

1N-61

43112

An Implementation of the Look-Ahead Lanczos Algorithm *p67* for Non-Hermitian Matrices Part II

Roland W. Freund and Noël M. Nachtigal

RIACS Technical Report 90.46

November 1990

(NASA-CR-188910) AN IMPLEMENTATION OF THE
LOOK-AHEAD LANCZOS ALGORITHM FOR
NON-HERMITIAN MATRICES, PART 2 (Research
Inst. for Advanced Computer Science) 67 p

N92-10312

CSCL 09B 63/61 0043112
Unclas

An Implementation of the Look-Ahead Lanczos Algorithm for Non-Hermitian Matrices Part II

Roland W. Freund and Noël M. Nachtigal

**The Research Institute for Advanced Computer Science is operated by
Universities Space Research Association (USRA),
The American City Building, Suite 311, Columbia, MD 21044, (301)730-2656.**

**Work reported herein was supported in part by DARPA via Cooperative
Agreement NCC 2-387 between NASA and USRA.**

An Implementation of the Look-Ahead Lanczos Algorithm for Non-Hermitian Matrices Part II

ROLAND W. FREUND

Research Institute for Advanced Computer Science

NOËL M. NACHTIGAL

Massachusetts Institute of Technology

In Part I [6] of this paper, we have presented an implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices. Here, we show how the look-ahead Lanczos process — combined with a quasi-minimal residual (QMR) approach — can be used to develop a robust black box solver for large sparse non-Hermitian linear systems. Details of an implementation of the resulting QMR algorithm are presented. It is demonstrated that the QMR method is closely related to the biconjugate gradient (BCG) algorithm; however, unlike BCG, the QMR algorithm has smooth convergence curves and good numerical properties. In particular, BCG iterates can be recovered stably from the QMR process. We report numerical experiments with our implementation of the look-ahead Lanczos algorithm, both for eigenvalue problems and linear systems. Also, program listings of FORTRAN implementations of the look-ahead Lanczos algorithm and the QMR method are included.

Categories and Subject Descriptors: G.1.3 [Numerical Analysis]: Numerical Linear Algebra—eigenvalues; *linear systems (direct and iterative methods)*; *sparse and very large systems*; G.4 [Mathematics of Computing]: Mathematical Software

General Terms: Large sparse linear systems and eigenvalue problems, iterative methods

Additional Key Words and Phrases: Non-Hermitian linear systems, quasi-minimal residual property, Lanczos method, biconjugate gradients

The work of R. W. Freund and N. M. Nachtigal was supported in part by DARPA via Cooperative Agreement NCC 2-387 between NASA and the Universities Space Research Association (USRA).

Authors' addresses: R. W. Freund, RIACS, Mail Stop Ellis Street, NASA Ames Research Center, Moffett Field, CA 94035, and Institut für Angewandte Mathematik, Universität Würzburg, D-W8700 Würzburg, Federal Republic of Germany; N. M. Nachtigal, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139.

8. INTRODUCTION

This report is a continuation of Part I [6] where we have presented an implementation of the look-ahead Lanczos algorithm for general non-Hermitian matrices A .

The purpose of the present paper is twofold. First, we show how the look-ahead Lanczos process — combined with a quasi-minimal residual (QMR hereafter) approach — can be used to solve large sparse non-Hermitian linear systems $Ax = b$. An implementation of the resulting QMR method is discussed in detail. The QMR approach was first proposed for the special case of complex symmetric matrices in [5]; for further properties of the method for general non-Hermitian matrices, we refer the reader to [7]. Second, we report numerical experiments with our implementation of the look-ahead Lanczos algorithm, both for eigenvalue problems and linear systems. Also, program listings of FORTRAN implementations of the look-ahead Lanczos algorithm and the QMR method are included.

Note that we continue the numbering of Part I [6] of this paper. Hence, the numbers 1, 2, ..., 7 refer to sections in Part I.

The outline of Part II is as follows. In Section 9, we describe the basic idea of the QMR approach. In Section 10, details of an actual implementation of the QMR algorithm are given. In Section 11, we discuss some of the properties of the QMR method. It is shown, for example, how iterates of the biconjugate gradient algorithm (BCG hereafter) can be obtained — when they exist — from the QMR algorithm. In Section 12, we briefly discuss how to incorporate preconditioning into the QMR method and describe two preconditioners which we have used for our numerical tests. In Section 13, numerical examples are presented. In Section 14, we make some concluding remarks. Finally, FORTRAN programs are listed in an Appendix.

Throughout this paper, A denotes a complex, in general non-Hermitian, $N \times N$ matrix. For given non-zero starting vectors, $v_1 \in \mathbb{C}^N$ and $w_1 \in \mathbb{C}^N$, we denote by \hat{v}_n and \hat{w}_n , $n = 1, 2, \dots$ the vectors (normalized to have unit length) generated by the look-ahead Lanczos method described in Part I [6]. Generally, we use the same notation as introduced in Sections 2 and 3. Hence

$$\hat{V}^{(n)} = [\hat{v}_1 \quad \hat{v}_2 \quad \dots \quad \hat{v}_n] = [\hat{V}_1 \quad \hat{V}_2 \quad \dots \quad \hat{V}_k] \quad (8.1)$$

and

$$\hat{W}^{(n)} = [\hat{w}_1 \quad \hat{w}_2 \quad \dots \quad \hat{w}_n] = [\hat{W}_1 \quad \hat{W}_2 \quad \dots \quad \hat{W}_k].$$

Here $k = k(n)$ is the number of the block containing the vector \hat{v}_n . Recall (cf. (2.1) and (2.19)) that the Lanczos vectors span the Krylov subspaces

$$K_n(v_1, A) := \text{span}\{v_1, Av_1, \dots, A^{n-1}v_1\},$$

$$K_n(w_1, A^T) := \text{span}\{w_1, A^T w_1, \dots, (A^T)^{n-1} w_1\},$$

i.e.,

$$K_n(v_1, A) = \left\{ \hat{V}^{(n)} z \mid z \in \mathbb{C}^n \right\}, \quad (8.2)$$

$$K_n(w_1, A^T) = \left\{ \hat{W}^{(n)} z \mid z \in \mathbb{C}^n \right\}. \quad (8.3)$$

By (2.20), left and right Lanczos vectors corresponding to different blocks are biorthogonal, i.e.,

$$\hat{W}_j^T \hat{V}_l = 0, \quad j \neq l. \quad (8.4)$$

Furthermore, in view of (3.7),

$$\hat{W}_l^T \hat{V}_l \text{ is nonsingular for complete blocks } l. \quad (8.5)$$

Finally, we will make use of the first relation in (3.3):

$$A \hat{V}^{(n)} = \hat{V}^{(n)} \hat{H}^{(n)} + [0 \quad \cdots \quad 0 \quad \rho_{n+1} \hat{v}_{n+1}]. \quad (8.6)$$

Here, by (3.5) and (3.6),

$$\hat{H}^{(n)} := \begin{bmatrix} \hat{\alpha}_1 & \hat{\beta}_2 & 0 & \cdots & 0 \\ \hat{\gamma}_2 & \hat{\alpha}_2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \hat{\beta}_k \\ 0 & \cdots & 0 & \hat{\gamma}_k & \hat{\alpha}_k \end{bmatrix} \quad (8.7)$$

is a $n \times n$ block tridiagonal matrix with diagonal and sub-diagonal blocks of the form

$$\hat{\alpha}_l = \begin{bmatrix} * & \cdots & \cdots & \cdots & * \\ \rho_{n_l+1} & \ddots & & & \vdots \\ 0 & \rho_{n_l+2} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \rho_{n_{l+1}-1} & * \end{bmatrix}, \quad \hat{\gamma}_l = \begin{bmatrix} 0 & \cdots & \cdots & 0 & \rho_{n_l} \\ \vdots & \ddots & & & 0 \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 \end{bmatrix}. \quad (8.8)$$

Moreover, we have set

$$\rho_j := \frac{s_j}{s_{j-1}} > 0, \quad j = 2, 3, \dots \quad (8.9)$$

Note that $\hat{H}^{(n)}$ and all the blocks $\hat{\alpha}_l$ are upper Hessenberg matrices with positive sub-diagonal elements.

9. THE QUASI-MINIMAL RESIDUAL ALGORITHM

We are interested in using the look-ahead Lanczos algorithm to solve linear systems

$$Ax = b. \quad (9.1)$$

Here, $b \in \mathbb{C}^N$ is some given right-hand side. Furthermore, it is always assumed that A is nonsingular.

Given any initial guess $x_0 \in \mathbb{C}^N$ for the exact solution $A^{-1}b$ of (9.1), we will construct iterates x_n , $n = 1, 2, \dots$, such that

$$x_n \in x_0 + K_n(r_0, A). \quad (9.2)$$

In the sequel, $r_n = b - Ax_n$ will denote the residual vector corresponding to the n th iterate x_n . We choose the initial residual $v_1 = r_0$ as one of the two starting vectors for the look-ahead Lanczos algorithm. Then, in view of (8.2), the right Lanczos vectors $\hat{v}_1, \hat{v}_2, \dots, \hat{v}_n$ span $K_n(r_0, A)$, and hence we have the parametrization

$$x_n = x_0 + \hat{V}^{(n)}z, \quad z \in \mathbb{C}^n, \quad (9.3)$$

for all possible iterates (9.2). Note that the second starting vector, $w_1 \in \mathbb{C}^N$, is still unspecified. Due to the lack of a criterion for the choice of w_1 , one usually sets $w_1 = r_0$ in practice.

Next, letting

$$\hat{H}_e^{(n)} := \begin{bmatrix} \hat{H}^{(n)} \\ \rho_{n+1} e_n^T \end{bmatrix}, \quad e_n^T := [0 \quad \dots \quad 0 \quad 1], \quad (9.4)$$

we can rewrite (8.6) as

$$A\hat{V}^{(n)} = \hat{V}^{(n+1)}\hat{H}_e^{(n)}. \quad (9.5)$$

By (9.5), the residual vectors corresponding to (9.3) satisfy

$$r_n = r_0 - A\hat{V}^{(n)}z = r_0 - \hat{V}^{(n+1)}\hat{H}_e^{(n)}z = \hat{V}^{(n+1)}(\tilde{\beta}^{(n)} - \hat{H}_e^{(n)}z), \quad (9.6)$$

where

$$\tilde{\beta}^{(n)} = [\rho_1 \quad 0 \quad \dots \quad 0]^T \in \mathbb{R}^{n+1} \quad \text{with} \quad \rho_1 = \|r_0\|.$$

We introduce an $(n+1) \times (n+1)$ diagonal matrix $\Omega^{(n)} = \text{diag}(\omega_1, \omega_2, \dots, \omega_{n+1})$, $\omega_i > 0$, to serve as a free parameter that can be used to modify the scaling of the problem. In our numerical experiments, the simplest scaling $\Omega^{(n)} = I_{n+1}$ gave satisfactory results.

However, better strategies for choosing $\Omega^{(n)}$ might be possible, and therefore we have formulated the QMR approach with a general scaling matrix $\Omega^{(n)}$. With it, (9.6) reads

$$\begin{aligned} r_n &= \hat{V}^{(n+1)} \left(\Omega^{(n)} \right)^{-1} \Omega^{(n)} \left(\tilde{\beta}^{(n)} - \hat{H}_e^{(n)} z \right) \\ &= \hat{V}^{(n+1)} \left(\Omega^{(n)} \right)^{-1} \left(\beta^{(n)} - \Omega^{(n)} \hat{H}_e^{(n)} z \right), \end{aligned} \quad (9.7)$$

where $\beta^{(n)} = \omega_1 \tilde{\beta}^{(n)}$.

Ideally, we would like to choose $z \in \mathbb{C}^n$ in (9.3) such that $\|r_n\|$ is minimal. However, since in general $\hat{V}^{(n+1)}$ is not unitary, this would require $\mathcal{O}(Nn^2)$ work, which is too expensive. We will instead minimize just the Euclidean norm of the bracketed terms, i.e., we will choose $z = z^{(n)} \in \mathbb{C}^n$ as the solution of the least squares problem

$$\left\| \beta^{(n)} - \Omega^{(n)} \hat{H}_e^{(n)} z^{(n)} \right\| = \min_{z \in \mathbb{C}^n} \left\| \beta^{(n)} - \Omega^{(n)} \hat{H}_e^{(n)} z \right\|. \quad (9.8)$$

Note that, by (8.7–9) and (9.4), $\hat{H}_e^{(n)}$ and $\Omega^{(n)} \hat{H}_e^{(n)}$ are $(n+1) \times n$ matrices with full column rank n . This guarantees that the solution $z^{(n)}$ of (9.8) is unique. Hence, (9.8) together with (9.3) define a unique n th iterate x_n . In view of the minimization property (9.8), we refer to this iteration scheme as the *quasi-minimal residual* (QMR) method.

For the solution of the least squares problem (9.8), we use the standard approach (see, e.g., [8, Chapter 6]) based on a QR decomposition of $\Omega^{(n)} \hat{H}_e^{(n)}$:

$$\left(Q^{(n)} \right)^H \begin{bmatrix} R^{(n)} \\ 0 \dots 0 \end{bmatrix} = \Omega^{(n)} \hat{H}_e^{(n)}. \quad (9.9)$$

Here, $Q^{(n)}$ is a unitary $(n+1) \times (n+1)$ matrix, and $R^{(n)}$ is a nonsingular upper triangular $n \times n$ matrix. Inserting (9.9) in (9.8) yields

$$\begin{aligned} \min_{z \in \mathbb{C}^n} \left\| \beta^{(n)} - \Omega^{(n)} \hat{H}_e^{(n)} z \right\| &= \min_{z \in \mathbb{C}^n} \left\| \beta^{(n)} - \left(Q^{(n)} \right)^H \begin{bmatrix} R^{(n)} \\ 0 \dots 0 \end{bmatrix} z \right\| \\ &= \min_{z \in \mathbb{C}^n} \left\| \left(Q^{(n)} \right)^H \left(Q^{(n)} \beta^{(n)} - \begin{bmatrix} R^{(n)} \\ 0 \dots 0 \end{bmatrix} z \right) \right\| \\ &= \min_{z \in \mathbb{C}^n} \left\| Q^{(n)} \beta^{(n)} - \begin{bmatrix} R^{(n)} \\ 0 \dots 0 \end{bmatrix} z \right\|. \end{aligned} \quad (9.10)$$

Hence, $z^{(n)}$ is given by

$$z^{(n)} = \left(R^{(n)}\right)^{-1} \left(t^{(n)}\right)_{1:n}, \quad \text{where} \quad t^{(n)} = \begin{bmatrix} \tau_1 \\ \vdots \\ \tau_n \\ \tilde{\tau}_{n+1} \end{bmatrix} = Q^{(n)} \beta^{(n)}. \quad (9.11)$$

Furthermore, we have

$$\left\| \beta^{(n)} - \Omega^{(n)} \hat{H}_\epsilon^{(n)} z^{(n)} \right\| = |\tilde{\tau}_{n+1}|. \quad (9.12)$$

10. IMPLEMENTATION DETAILS

In this section, we describe in detail the actual implementation of the QMR algorithm. We break the discussion in two logical blocks: first, updating the QR decomposition of $\Omega^{(n)}\hat{H}_e^{(n)}$, and second, updating the QMR iterates x_n . Many of the details of updating the QR decomposition of $\Omega^{(n)}\hat{H}_e^{(n)}$ can also be found elsewhere (see, *e.g.*, [8, Chapter 12]); they are included here for completeness. Furthermore, we remark that the approach for updating the iterates x_n is based on a technique which was first used by Paige and Saunders [11] in connection with their SYMMLQ and MINRES algorithms for real symmetric matrices.

The QR decomposition (9.9) of $\Omega^{(n)}\hat{H}_e^{(n)}$ is computed by means of Givens rotations, taking advantage of the fact that $\Omega^{(n)}\hat{H}_e^{(n)}$ is an upper Hessenberg matrix. We use Givens rotations of the form

$$G_n = \begin{bmatrix} I_{n-1} & 0 & 0 \\ 0 & c_n & s_n \\ 0 & -\overline{s_n} & c_n \end{bmatrix}, \quad \text{with } c_n \in \mathbb{R}, s_n \in \mathbb{C}, c_n^2 + |s_n|^2 = 1. \quad (10.1)$$

Let

$$h = \begin{bmatrix} x \\ \vdots \\ x \\ a \\ b \end{bmatrix} \in \mathbb{C}^{n+1}, \quad \begin{bmatrix} a \\ b \end{bmatrix} \neq 0,$$

be a given vector. Then, by choosing

$$\begin{aligned} c_n &= \frac{|a|}{\sqrt{|a|^2 + |b|^2}}, \quad \overline{s_n} = c_n \frac{b}{a}, & \text{if } a \neq 0, \\ c_n &= 0, \quad \overline{s_n} = 1, & \text{if } a = 0, \end{aligned} \quad (10.2)$$

in (10.1), we obtain a Givens rotation G_n which zeroes out the last element of h

$$G_n \begin{bmatrix} x \\ \vdots \\ x \\ a \\ b \end{bmatrix} = \begin{bmatrix} x \\ \vdots \\ x \\ * \\ 0 \end{bmatrix}.$$

Here 'x' denotes elements that are left unchanged by the rotation. Clearly, G_n is unitary; furthermore, products of Givens rotations are also unitary. In particular,

$$Q^{(n)} = G_n \begin{bmatrix} G_{n-1} & 0 \\ 0 & 1 \end{bmatrix} \cdots \begin{bmatrix} G_1 & 0 \\ 0 & I_{n-1} \end{bmatrix} \quad (10.3)$$

is unitary. In general, then, the QR decomposition of $\Omega^{(n)} \hat{H}_e^{(n)}$ can be computed as follows. After the n th column of $\hat{H}_e^{(n)}$ is built, one first premultiplies it by $\Omega^{(n)}$, and then by all the previous Givens rotations, thus computing

$$h = G_{n-1} \begin{bmatrix} G_{n-2} & 0 \\ 0 & 1 \end{bmatrix} \cdots \begin{bmatrix} G_1 & 0 \\ 0 & I_{n-1} \end{bmatrix} \left(\Omega^{(n)} \hat{H}_e^{(n)} \right)_{:,n+1}.$$

One then computes G_n from (10.2) with $a = h_n$, $b = h_{n+1}$ ($= \omega_{n+1} \rho_{n+1}$), and finally obtains the n th column of $R^{(n)}$ by applying G_n to h . For later use, we notice that

$$\left| s_n \left(R^{(n)} \right)_{n,n} \right| = \omega_{n+1} \rho_{n+1} \quad (10.4)$$

which is readily verified by means of (10.2).

In our case, $\Omega^{(n)} \hat{H}_e^{(n)}$ is also block tridiagonal, which means that not all the previous Givens rotations have to be applied. Let n_G denote the index of the first Givens rotation that has to be applied. Then

$$n_G = \begin{cases} \max(n_{k-1} - 1, 1) & \text{if } \hat{v}_n \text{ is an inner vector,} \\ \max(n_{k-2} - 1, 1) & \text{if } \hat{v}_n \text{ is a regular vector.} \end{cases}$$

Finally, note that if $n_G > 1$, then applying G_{n_G} to $\left(\Omega^{(n)} \hat{H}_e^{(n)} \right)_{:,n+1}$ will introduce a non-zero element in the n_G position.

Once the QR decomposition is updated, one updates the vector $t^{(n)}$ in (9.11) by setting

$$t^{(n)} = G_n \begin{bmatrix} t^{(n-1)} \\ 0 \end{bmatrix}. \quad (10.5)$$

Clearly, by (10.1), $t^{(n)}$ differs from $t^{(n-1)}$ only in its last two entries which are given by

$$\tau_n = c_n \tilde{\tau}_n \quad \text{and} \quad \tilde{\tau}_{n+1} = -\bar{s}_n \tilde{\tau}_n. \quad (10.6)$$

Next, we show that this newly introduced element $\tilde{\tau}_{n+1}$ can be used to obtain an upper bound for the norm of the residual r_n . First, note that, in view of (9.7),

$$\begin{aligned} \|r_n\| &= \left\| \hat{V}^{(n+1)} \left(\Omega^{(n)} \right)^{-1} \left(\beta^{(n)} - \Omega^{(n)} \hat{H}_e^{(n)} z^{(n)} \right) \right\| \\ &\leq \left\| \hat{V}^{(n+1)} \right\| \left\| \left(\Omega^{(n)} \right)^{-1} \right\| \left\| \beta^{(n)} - \Omega^{(n)} \hat{H}_e^{(n)} z^{(n)} \right\|. \end{aligned} \quad (10.7)$$

Now

$$\|\hat{V}^{(n+1)}\| \leq \sqrt{n+1}, \quad (10.8)$$

since $\hat{V}^{(n+1)}$ has $n+1$ columns of Euclidean norm 1, and

$$\left\| \left(\Omega^{(n)} \right)^{-1} \right\| = \max_i \left(\frac{1}{\omega_i} \right), \quad i = 1, \dots, n+1, \quad (10.9)$$

as $\Omega^{(n)}$ is diagonal. Combining (10.7-9) and (9.12), we get

$$\|r_n\| \leq \sqrt{n+1} |\tilde{r}_{n+1}| \max_i \left(\frac{1}{\omega_i} \right). \quad (10.10)$$

Let us now turn to updating the iterates x_n . We will update x_n only when the current block k is closed, i.e., when

$$n = n_{k+1} - 1. \quad (10.11)$$

This means that the QR decomposition of $\Omega^{(n)} \hat{H}_e^{(n)}$ has been updated as described above, and that $(t^{(n)})_{1:n}$ will hereafter remain unchanged. Inserting (9.11) in (9.3) yields

$$\begin{aligned} x_n &= x_0 + \hat{V}^{(n)} z^{(n)} = x_0 + \hat{V}^{(n)} \left(R^{(n)} \right)^{-1} \left(t^{(n)} \right)_{1:n} \\ &= x_0 + P^{(n)} \left(t^{(n)} \right)_{1:n} \end{aligned} \quad (10.12)$$

where we have introduced the matrix of direction vectors

$$P^{(n)} = \hat{V}^{(n)} \left(R^{(n)} \right)^{-1} = [P_1 \ P_2 \ \dots \ P_k]. \quad (10.13)$$

Here the partitioning of $P^{(n)}$ into submatrices on the right-hand side of (10.13) is similar to (8.1): the matrices P_l contain the direction vectors corresponding to block l , $l = 1, 2, \dots, k$. Then, with (10.11-13), we arrive at the update formula

$$x_n = x_{n_k-1} + P_k \left(t^{(n)} \right)_{n_k:n}, \quad n = n_{k+1} - 1. \quad (10.14)$$

It remains to show how to compute P_k . To this end, we first note that

$$\hat{V}^{(n)} = P^{(n)} R^{(n)} = P^{(n)} \left[\begin{array}{ccc|ccc} & & & & 0 & \dots & 0 \\ & & & & \vdots & & \vdots \\ & & & & 0 & \dots & 0 \\ & & & & & Z_k & \\ & & & & & Y_k & \\ \hline 0 & \dots & \dots & 0 & & & \\ \vdots & & & \vdots & & & \\ 0 & \dots & \dots & 0 & & R_k & \end{array} \right], \quad (10.15)$$

where Z_k is a $1 \times h_k$ block, Y_k is a $h_{k-1} \times h_k$ block, and R_k is a $h_k \times h_k$ block. Recall that $h_{k-1} = n_k - n_{k-1}$ and $h_k = n_{k+1} - n_k$ denote the sizes of blocks $k-1$ and k , respectively. The block structure of $R^{(n)}$ in (10.15) is a consequence of the block structure (8.7) of $\hat{H}^{(n)}$ and of the fact that the fill-in from the QR decomposition of $\Omega^{(n)} \hat{H}_e^{(n)}$ is limited to the row above each block $\hat{\beta}_l$, $l = 2, 3, \dots, k$.

To save work, we compute and store the vectors $P_k R_k$, as opposed to computing and storing P_k . Rewriting (10.15) in terms of the stored vectors, we obtain the update formula

$$P_k R_k = \hat{V}_k - (P_{k-2} R_{k-2}) (R_{k-2})^{-1} \begin{bmatrix} 0 & \cdots & 0 \\ & Z_k & \end{bmatrix} - (P_{k-1} R_{k-1}) (R_{k-1})^{-1} Y_k. \quad (10.16)$$

Moreover, instead of (10.14), we actually use the update formula

$$x_n = x_{n_{k-1}} + (P_k R_k) (R_k)^{-1} \left(t^{(n)} \right)_{n_k:n}, \quad n = n_{k+1} - 1.$$

Finally, a note about computing the middle term in (10.16). If one computes in order from left to right, i.e.,

$$\left((P_{k-2} R_{k-2}) (R_{k-2})^{-1} \right) \begin{bmatrix} 0 & \cdots & 0 \\ & Z_k & \end{bmatrix},$$

then the work involved is $\mathcal{O}(N h_{k-2} + N h_k)$. If one computes in order from right to left, i.e.,

$$(P_{k-2} R_{k-2}) \left((R_{k-2})^{-1} \begin{bmatrix} 0 & \cdots & 0 \\ & Z_k & \end{bmatrix} \right),$$

then the work involved is $\mathcal{O}(h_k h_{k-2} + N h_k h_{k-2})$. This means that if $h_k = 1$ or $h_{k-2} = 1$, it is cheaper to multiply from right to left; otherwise, it is cheaper to multiply from left to right. In both cases, one takes advantage of the fact that Z_k is a row vector, which means that only the last column of the matrix premultiplying Z_k has to be computed. In particular, only the last column of $(R_{k-2})^{-1}$ is needed and has to be stored.

11. FURTHER PROPERTIES OF THE QMR ALGORITHM

In this section, we derive an update formula for the QMR residual vectors. Moreover, it is shown that BCG iterates can be easily recovered from the QMR process.

In the QMR algorithm described in the last section, neither the residual vectors r_n nor their norms $\|r_n\|$ are generated explicitly. Instead, the upper bound (10.10) for $\|r_n\|$ is available. In our implementation, we monitor this upper bound and switch over to computing the true residual vector and its norm only in the last few iteration steps. Another possibility would be to update the residual vector in each step. The following proposition shows that this can be done at the cost of one additional SAXPY per step.

Proposition 1. For $n = 1, 2, \dots$:

$$r_n = |s_n|^2 r_{n-1} + \frac{c_n \tilde{r}_{n+1}}{\omega_{n+1}} \hat{v}_{n+1}. \quad (11.1)$$

Proof. By inserting $z = z^{(n)}$ from (9.11) in (9.7) and using (9.9), we obtain

$$r_n = \tilde{r}_{n+1} y_{n+1} \quad \text{where} \quad y_{n+1} = \hat{V}^{(n+1)} (\Omega^{(n)})^{-1} (Q^{(n)})^H \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}. \quad (11.2)$$

Now note that, by (10.3) and (10.1),

$$(Q^{(n)})^H = \begin{bmatrix} (Q^{(n-1)})^H & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} I_{n-1} & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \\ 0 & \dots & 0 \\ 0 & \dots & 0 \end{bmatrix} \begin{bmatrix} c_n & -s_n \\ \frac{c_n}{s_n} & c_n \end{bmatrix}. \quad (11.3)$$

By means of (11.3), one readily verifies that two successive vectors y_{n+1} and y_n in (11.2) are connected by

$$y_{n+1} = -s_n y_n + \frac{c_n}{\omega_{n+1}} \hat{v}_{n+1}. \quad (11.4)$$

Finally, by inserting (11.4) in (11.2) and using the second relation in (10.6), we obtain (11.1). \square

Next we turn to the BCG method [9] and show its connection with the QMR algorithm. In order to distinguish quantities from the two approaches, superscripts *BCG* and *QMR* will be used. In the sequel, it is always assumed that BCG and QMR are started with the same initial guess x_0 .

In the BCG approach, one aims at computing iterates x_n^{BCG} which are characterized by the Galerkin type condition

$$w^T(b - Ax_n^{BCG}) = 0 \quad \text{for all } w \in K_n(w_1, A^T), \quad x_n^{BCG} \in x_0 + K_n(r_0, A). \quad (11.5)$$

(see, *e.g.*, [13]). Unfortunately, such an iterate need not exist for every n . This is one of the two sources for possible breakdowns which can occur in the classical BCG algorithm.

Next, we rewrite the condition (11.5) in terms of quantities generated by the look-ahead Lanczos algorithm. For this purpose, the same parametrization as in (9.3) is used:

$$x_n^{BCG} = x_0 + \hat{V}^{(n)}u^{(n)}, \quad u^{(n)} \in \mathbb{C}^n.$$

Then, in view of (8.6), the corresponding residual vector satisfies

$$\begin{aligned} r_n^{BCG} &= b - Ax_n^{BCG} = r_0 - A\hat{V}^{(n)}u^{(n)} \\ &= \hat{V}^{(n)} \left(\gamma^{(n)} - \hat{H}^{(n)}u^{(n)} \right) - \rho_{n+1} \left(u^{(n)} \right)_n \hat{v}_{n+1}, \\ \text{where } \gamma^{(n)} &= [\rho_1 \quad 0 \quad \cdots \quad 0]^T \in \mathbb{R}^n, \quad \rho_1 = \|r_0\|. \end{aligned} \quad (11.6)$$

Next, inserting (11.6) in (11.5) and using (8.3), we can rewrite (11.5) as follow:

$$\begin{aligned} \left(\hat{W}^{(n)} \right)^T \hat{V}^{(n)} \left(\gamma^{(n)} - \hat{H}^{(n)}u^{(n)} \right) &= \rho_{n+1} \left(u^{(n)} \right)_n \left(\hat{W}^{(n)} \right)^T \hat{v}_{n+1}, \\ x_n^{BCG} &= x_0 + \hat{V}^{(n)}u^{(n)}. \end{aligned} \quad (11.7)$$

In analogy to the QMR algorithm, we will attempt to recover the BCG iterate only when the current block k is closed. Thus, in the sequel, it is always assumed that $n = n_{k+1} - 1$, (cf. (10.11)). This guarantees that, by (8.5) and (8.4),

$$\left(\hat{W}^{(n)} \right)^T \hat{V}^{(n)} \text{ is nonsingular and } \left(\hat{W}^{(n)} \right)^T \hat{v}_{n+1} = 0, \quad (11.8)$$

respectively. Finally, with (11.8), the equation (11.7) reduces to

$$\hat{H}^{(n)}u^{(n)} = \gamma^{(n)}. \quad (11.9)$$

By means of (11.9), we can now derive a simple criterion for the existence of the n th BCG iterate. In the following proposition, c_j and s_j , $j = 1, \dots, n$, denote the elements of the j th Givens rotation used in our implementation of the QMR method, as described in Section 10. Moreover, p_n is the last column of the matrix $P^{(n)}$ introduced in (10.13).

Proposition 2. Let $n = n_{k+1} - 1$, $k = 0, 1, \dots$. Then the following are equivalent:

- (i) the BCG iterate x_n^{BCG} defined by (11.5) exists;
- (ii) $\hat{H}^{(n)}$ is nonsingular;
- (iii) $c_n \neq 0$.

Moreover, if x_n^{BCG} exists, then

$$x_n^{BCG} = x_n^{QMR} + \frac{\tau_n |s_n|^2}{c_n^2} p_n, \quad (11.10)$$

$$r_n^{BCG} = -\rho_{n+1} \left(u^{(n)} \right)_n \hat{v}_{n+1}, \quad (11.11)$$

$$\|r_n^{BCG}\| = \|r_0\| \cdot |s_1 s_2 \cdots s_{n-1} s_n| \frac{\omega_1}{\omega_{n+1} c_n}. \quad (11.12)$$

Proof. Clearly, an n th BCG iterate exists if, and only if, the linear system (11.9) has a solution. Recall that, by (11.6), $\gamma^{(n)}$ is a nonzero multiple of the first unit vector, and that $\hat{H}^{(n)}$ is an upper Hessenberg matrix whose subdiagonal elements are all nonzero (by (8.9)). Hence, the extended coefficient matrix $[\gamma^{(n)} \quad \hat{H}^{(n)}]$ of (11.9) has full row rank n . Consequently, (11.9) has a solution if, and only if, $\hat{H}^{(n)}$ is nonsingular. This shows the equivalence of (i) and (ii).

Next, using (9.9), (9.4), (10.1), and (10.3), one readily verifies that

$$Q^{(n-1)} \Omega^{(n-1)} \hat{H}^{(n)} = \begin{bmatrix} I_{n-1} & 0 \\ 0 & c_n \end{bmatrix} R^{(n)}. \quad (11.13)$$

This relation implies that (ii) and (iii) are equivalent.

Now assume $c_n \neq 0$. With (11.9) and (11.13), it follows that

$$\begin{aligned} x_n^{BCG} &= x_0 + \hat{V}^{(n)} u^{(n)}, \\ u^{(n)} &= \left(R^{(n)} \right)^{-1} \begin{bmatrix} I_{n-1} & 0 \\ 0 & 1/c_n \end{bmatrix} Q^{(n-1)} \Omega^{(n-1)} \gamma^{(n)}. \end{aligned} \quad (11.14)$$

Recalling the definitions of $\gamma^{(n)}$ and $\beta^{(n)}$ in (11.6) and (9.6), respectively, and using (9.11), we can rewrite (11.14) as follows:

$$x_n^{BCG} = x_0 + \hat{V}^{(n)} u^{(n)} \quad \text{where} \quad u^{(n)} = \left(R^{(n)} \right)^{-1} \begin{bmatrix} (t^{(n)})_{1:n-1} \\ \tilde{\tau}_n / c_n \end{bmatrix}. \quad (11.15)$$

By comparing (11.15) with (10.12), we obtain the relation

$$x_n^{BCG} = x_n^{QMR} + \left(\frac{\tilde{\tau}_n}{c_n} - \tau_n \right) p_n$$

which, in view of (10.6), is just (11.10). Equation (11.11) follows by inserting (11.9) in (11.6).

Finally, from (11.11) and (11.15), we deduce that

$$\|r_n^{BCG}\| = \frac{\rho_{n+1}}{c_n} \left| \frac{\tilde{r}_n}{\xi_n} \right| \quad \text{where} \quad \xi_n = \left(R^{(n)} \right)_{n,n}. \quad (11.16)$$

Note that, in view of (10.4),

$$\rho_{n+1} = \frac{|s_n \xi_n|}{\omega_{n+1}}. \quad (11.17)$$

Moreover, by (10.6) and since $\tilde{r}_1 = \omega_1 \|r_0\|$,

$$|\tilde{r}_n| = \omega_1 \|r_0\| \cdot |s_1 s_2 \cdots s_{n-1}|. \quad (11.18)$$

By inserting (11.17) and (11.18) in (11.16), we get (11.12), and this concludes the proof. \square

12. PRECONDITIONING

As for other conjugate gradient type methods, for solving realistic problems, it is crucial to combine the QMR algorithm with an efficient preconditioning technique. In this section, we show how to incorporate preconditioners into the QMR code. Also, we briefly describe two preconditioning techniques.

Let us first treat the problem in some generality. Suppose that we replace the original linear system (9.1), $Ax = b$, by an equivalent linear system

$$\tilde{A}y = \tilde{b}, \quad (12.1)$$

where \tilde{A} , y , and \tilde{b} are defined suitably to guarantee the equivalence between the two systems. Typically,

$$\tilde{A} = f_A(A), \quad y = f_x(x), \quad \tilde{b} = f_b(b)$$

for some functions f_A , f_x , and f_b . The goal in replacing (9.1) with (12.1) is to replace a "hard" system by an "easier" system. In particular, one attempts to ensure that the system (12.1) is easier to solve than (9.1), which translates into having a matrix \tilde{A} which is more suitable for some solver than A is, and having a function f_x which is easily invertible. The QMR algorithm for the new system (12.1) is then as follows:

Given an initial guess x_0 and a nonzero vector w_1 , set

$$\begin{aligned} y_0 &= f_x(x_0), \\ \tilde{r}_0 &= \tilde{b} - \tilde{A}y_0, \\ \rho_1 &= \|\tilde{r}_0\|, \\ \hat{v}_1 &= \frac{1}{\rho_1} \tilde{r}_0. \end{aligned}$$

Iterate with \tilde{A} to compute'

$$\begin{aligned} \tilde{A}\hat{V}^{(n)} &= \hat{V}^{(n+1)}\hat{H}_e^{(n)}, \\ \tilde{A}^T\hat{W}^{(n)} &= \hat{W}^{(n+1)}\hat{H}_e^{(n)}. \end{aligned}$$

When appropriate, compute the iterates

$$y_n = y_0 + \hat{V}^{(n)}z_n$$

where, given

$$\begin{aligned} \tilde{r}_n &= \tilde{b} - \tilde{A}y_n = \tilde{b} - \tilde{A}y_0 - \tilde{A}\hat{V}^{(n)}z_n = \tilde{r}_0 - \tilde{A}\hat{V}^{(n)}z_n \\ &= \hat{V}^{(n+1)}\left(\Omega^{(n+1)}\right)^{-1} \left(\begin{bmatrix} \omega_1\rho_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} - \Omega^{(n+1)}\hat{H}_e^{(n)}z_n \right), \end{aligned}$$

z_n is chosen to minimize

$$\left\| \begin{bmatrix} \omega_1 \rho_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} - \Omega^{(n+1)} \hat{H}_e^{(n)} z_n \right\|.$$

When appropriate, compute the iterates

$$x_n = f_x^{-1}(y_n)$$

Note that the term being minimized is part of \tilde{r}_n ; one could attempt to correlate this with r_n .

One approach in preconditioning is the following. Given a preconditioner matrix M , with $M \approx A$ in some sense, one uses a decomposition of M into

$$M = M_1 M_2$$

to precondition the original system $Ax = b$, by setting

$$\tilde{A} = M_1^{-1} A M_2^{-1}, \quad y = M_2(x - x_0), \quad \tilde{b} = M_1^{-1}(b - A x_0). \quad (12.2)$$

Clearly, with (12.2), (12.1) is equivalent to the original system (9.1). Furthermore, given iterates y_n generated for (12.1), one can recover x_n and r_n for (9.1) through

$$x_n = x_0 + M_2^{-1} y_n, \quad r_n = M_1 \tilde{r}_n,$$

with $y_0 = 0$ and $\tilde{r}_0 = M_1^{-1} r_0$. By setting $M_1 = I$ or $M_2 = I$, one obtains right or left preconditioning, respectively, in which cases the above formulas simplify somewhat. In general, however, for the QMR algorithm applied to a preconditioned system, one has to be able to compute $M_1^{-1} z$, $M_1^{-T} z$, $M_2^{-1} z$, and $M_2^{-T} z$, for arbitrary vectors z .

Two examples of preconditioners that fall in the category described above are the SSOR and the ILUT preconditioners. We briefly discuss them and our particular implementations.

• SSOR

The SSOR preconditioner is based on a decomposition of the matrix A into a nonsingular diagonal matrix D , a strictly lower triangular matrix L , and a strictly upper triangular matrix U , such that

$$A = D + L + U.$$

Note that D might have to be block diagonal to ensure it is nonsingular. For this preconditioner, one sets $M_1 = I$ or $M_2 = I$, depending on whether right or left preconditioning is desired. The preconditioner matrix M is given by

$$M_{SSOR} = (D + \omega L) D^{-1} (D + \omega U),$$

which gives

$$M_{SSOR}^{-1} = (D + \omega U)^{-1} D (D + \omega L)^{-1},$$

for some parameter ω . We take $\omega = 1$.

- **ILUT(k)**

The Incomplete LU decomposition is based on the LU decomposition of the coefficient matrix A . The full LU decomposition of A would result in factors L and U which, in general, have far more nonzero elements than A . The incomplete LU factorization aims to reduce this additional fill-in in the factors L and U .

In ILUT(k), we use the following strategy for dropping non-zero elements which would fill-in L and U . Each row of L and U is constructed subject to the restriction that only a small amount of fill-in, k more elements for each, is allowed beyond the number of elements of A already present in that row (in the lower and upper part, respectively). Furthermore, elements which are deemed to make only an insignificant contribution to the decomposition are also dropped. For example, this means that if $LENL$ is the maximum number of elements allowed for some row of L , K is the actual number of elements of that row computed by the elimination process, and TOL is the cutoff tolerance, then the algorithm orders the K elements in decreasing order of magnitude, and keeps only up to $\min(K, LENL)$ elements, or until the elements reach the level TOL , whichever cutoff comes first. The resulting matrices L and U are then used as $M_1 = L$, $M_2 = U$.

Note that the variant of ILU used is different from the standard one. For a symmetric matrix A , the standard ILU preconditioner [10] preserves the sparsity structure of the matrix, i.e., for $k = 0$, the preconditioner matrices have non-zero elements only in those locations where A itself has non-zero elements. In [10] it is shown that this strategy does produce a good preconditioner, provided that A is a symmetric M -matrix. For a general nonsymmetric matrix, there is no reason to preserve the sparsity structure of A . Since we are mainly concerned with nonsymmetric matrices, our implementation discards elements subject only to the constraints of fill-in and size, without regard to the sparsity structure of A . However, this does mean that if A is symmetric, we do not recover the standard ILU preconditioner.

13. NUMERICAL EXPERIMENTS

In this section, we present some numerical examples obtained with our FORTRAN code. We show both eigenvalue problems, which involve only the Lanczos algorithm, and non-symmetric linear systems, which also involve the QMR algorithm. We also indicate how many blocks of size bigger than 2 the algorithm built during each run. We only report blocks that were successfully closed; blocks that were rebuilt after updating the estimate for the norm of the matrix are not counted. Unless otherwise indicated, the elements of the starting vectors v_1 and w_1 were random numbers from a normal distribution with mean 0.0 and variance 1.0. Also, in all cases, the user-supplied estimate for the norm of the matrix was set to 1, thus forcing the algorithm to estimate the norm on its own. For eigenvalue problems, we use the heuristic presented in [2] to identify and eliminate spurious eigenvalues. In the plots, we show the true eigenvalues of the matrix (denoted by '.'), as compared to the computed Lanczos eigenvalues (denoted by 'o'). For the linear systems, the starting guess x_0 was always zero and the convergence requirement was a reduction by a factor of 10^{-6} in the norm of the residual. We always used the QMR algorithm with no scaling, i.e., $\Omega^{(n)} \equiv I_{n+1}$ in (9.7), and we preconditioned the systems as discussed in Section 12. For the SSOR preconditioners, we always used $\omega = 1.0$; for the ILUT preconditioners, we used ILUT(0) and ILUT(4), with the threshold set to $TOL = 0.001$. While we only show results for the right SSOR preconditioner, we also ran the left SSOR preconditioner, and in all cases, the number of iterations needed to converge was roughly the same as for the right SSOR preconditioner; we prefer the right preconditioner because its residual vector is identical to the residual vector of the unpreconditioned system. In the plots, we show two convergence curves: the top curve (solid) is the upper bound (10.10) used in the algorithm for the residual norm, while the lower curve (dotted) is the computed residual norm. In practice, one would monitor the upper bound until it reached the convergence range, and then possibly switch to computing the true residual norm until convergence. The vertical scale is the same on all the convergence plots, but the horizontal scale usually changes from one plot to the next. Finally, unless otherwise indicated, all the examples were run on a Sun Sparc Station 1, in double precision.

Example 1. The first example is taken from [1]. We include it here to make the point that, as indicated in Section 2, singular and deficient polynomials do indeed occur. We have

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \quad v_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad w_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \end{bmatrix}.$$

With these starting vectors, the algorithm builds a 2×2 block, followed by a 4×4 block, followed by 1×1 blocks. The 2nd and 4th degree polynomials are singular, while the

5th, and 6th degree polynomials are deficient. After 8 steps, the algorithm finds invariant subspaces with respect to both A and A^T , and the computed eigenvalues match the true eigenvalues exactly, as shown in Figure 1.

Example 2. This example is an eigenvalue problem, taken from [12], whose exact eigenvalues are known. Generally, problems of this type arise in modeling concentration waves in reaction and transport interaction of chemical solutions in a tubular reactor. The particular test problem used here corresponds to the so-called Brusselator wave model. We took $N = 200$ and computed eigenvalues after 20 (Figure 2) and after 100 (Figure 3) Lanczos steps. For the latter case, the algorithm built 2 blocks of size 2. As expected, the extreme eigenvalues converge rapidly, while the interior ones require more iterations. For this example, the Lanczos algorithm computes all the eigenvalues after 200 steps; however, it requires as much work as the direct computation of the eigenvalues would require.

Example 3. This example comes from a problem in hydrodynamics. The aim is to model a 2-D wave sloshing in a tank using integral equations and Green's theorem. The approach used is to transform the elliptic mixed boundary-value problem into boundary integrals, leading to mixed first and second kind equations, and then to discretize the integral equations. This is done at every time step. It yields a dense nonsymmetric matrix which is used to solve a system for the velocity at the boundary points. Once the velocity is known, the free boundary is then advanced in time. The routine to generate the matrix was provided by Dick Yue and Hongbo Xu from the MIT Department of Ocean Engineering. We used the matrix as it arises at a time when the wave is beginning to overturn (see Figure 4); the right and bottom walls are fixed, the left wall is a wave-maker which produced the wave. In Figure 5, we show the convergence results for $N = 640$, with the system preconditioned with the right SSOR preconditioner; the algorithm built 2 blocks of size 2. In this example, the right-hand side vector was prescribed by the physics in the problem.

Example 4. This example is taken from [3]. The following partial differential equation models heat conduction in a box, where a side is heated and the flow is rotating:

$$\begin{aligned} -\Delta u + [v_x \ v_y \ v_z] \cdot (\nabla u) &= 0 \quad \text{on } (-1, 1) \times (-1, 1) \times (-1, 1), \\ \text{with Dirichlet boundary conditions } u &= \begin{cases} 100 & \text{if } z = -1, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \quad (13.1)$$

Here,

$$\begin{aligned} v_x &= C_p C_0 y z (1 - x^2)^2 (1 - y^2) (1 - z^2), \\ v_y &= -C_p C_1 x z (1 - y^2)^2 (1 - x^2) (1 - z^2), \\ v_z &= -C_p C_1 x y (1 - z^2)^2 (1 - x^2) (1 - y^2), \\ C_0 &= 27/2, \quad C_1 = C_0/2, \end{aligned}$$

and $C_p > 0$ is a free parameter. We discretize (13.1) using centered differences on a uniform $m \times m \times m$ grid with mesh size $h = 2/(m + 1)$. The resulting linear system has a sparse coefficient matrix A of order $N = m^3$. For our experiments, we have chosen the parameter $C_p = 1/h$. Note that this choice guarantees that the cell Reynolds number is smaller than one, and hence centered differences yield a stable discretization of (13.1). In this and all subsequent sparse examples, we have relied heavily on routines from SPARSKIT [14] for the basic matrix operations. In Figure 6, we show the convergence results for $m = 31$ ($N = 29791$), with the system preconditioned with the right SSOR preconditioner; the algorithm built only 1×1 blocks. The right-hand side vector was chosen to describe a physically possible solution. This example was run on a Cray-2 at the NASA Ames Research Center.

Example 5 and 6. Finally, Examples 5 and 6 were taken from the Harwell-Boeing sparse matrix collection [4]. On these examples, we ran both the SSOR preconditioners, and two versions of the ILUT preconditioner, namely ILUT(0) and ILUT(4). Example 5 is the first matrix from the OILGEN collection, called ORSREG 1. It comes from an oil reservoir simulation on a $21 \times 21 \times 5$ full grid; the order of the matrix is 2205, and it has 14133 non-zero elements. In Figure 7, we show the convergence curves for the right SSOR preconditioner, in Figure 8, the convergence curves for ILUT(0), and in Figure 9, the convergence curves for ILUT(4). The algorithm built one block of size 2 for the SSOR preconditioner, one block of size 2 and one of size 3 for the ILUT(0) case, and no blocks of size bigger than 1 for the ILUT(4) case. Example 6 is the fifth matrix from the SHERMAN collection, called SHERMAN 5. It comes from a fully implicit black oil simulator on a $16 \times 23 \times 3$ grid, with three unknowns. The order of the matrix is 3312, and it has 20793 non-zero elements. In Figure 10, we show the convergence curves for the right SSOR preconditioner, in Figure 11, the convergence curves for ILUT(0), and in Figure 12, the convergence curves for ILUT(4). For this case, the convergence curves for the two ILUT preconditioners are almost identical. The algorithm built one block of size 2 for the SSOR preconditioner, three blocks of size 2 for the ILUT(0) case, and one block of size 2 for the ILUT(4) case.

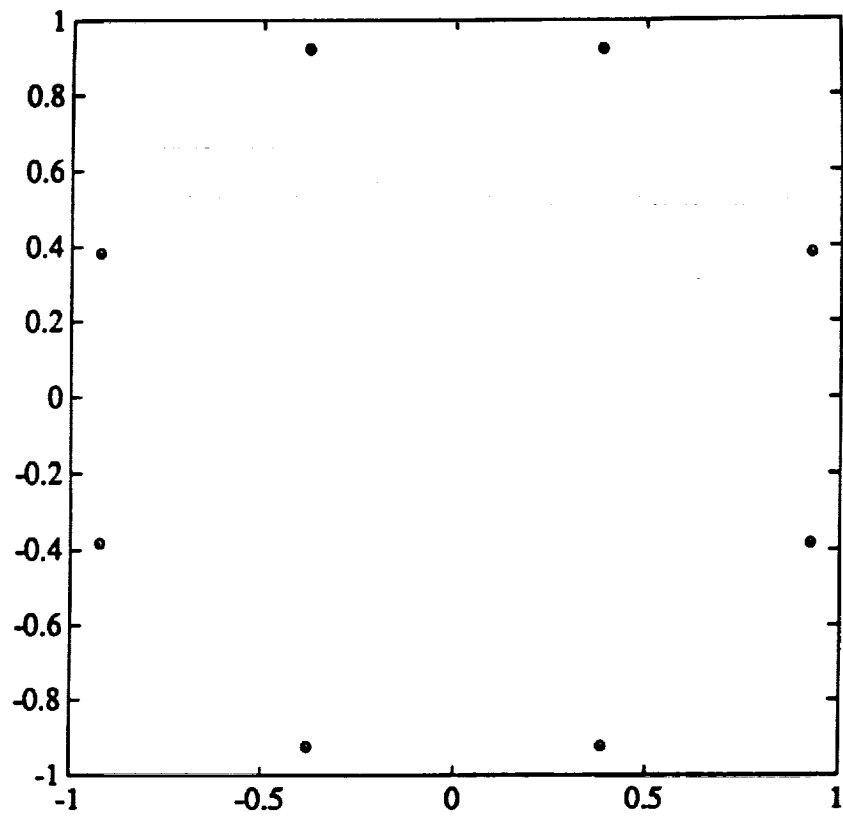


Figure 1. Eigenvalues of Example 1.

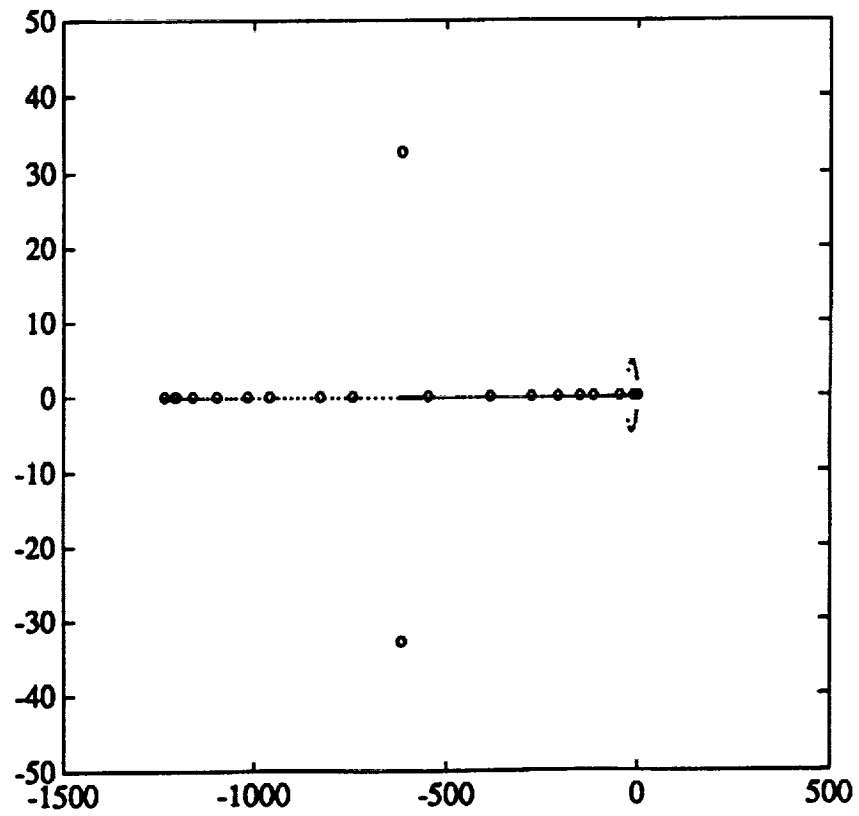


Figure 2. Eigenvalues of the Brusselator example after 20 steps.

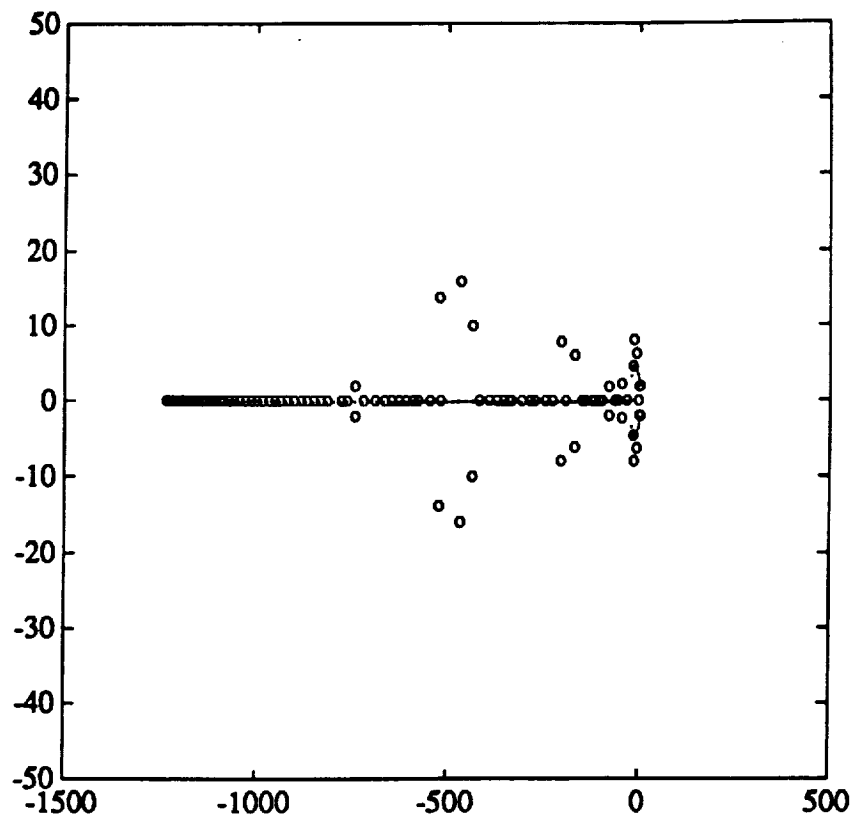


Figure 3. Eigenvalues of the Brusselator example after 100 steps.

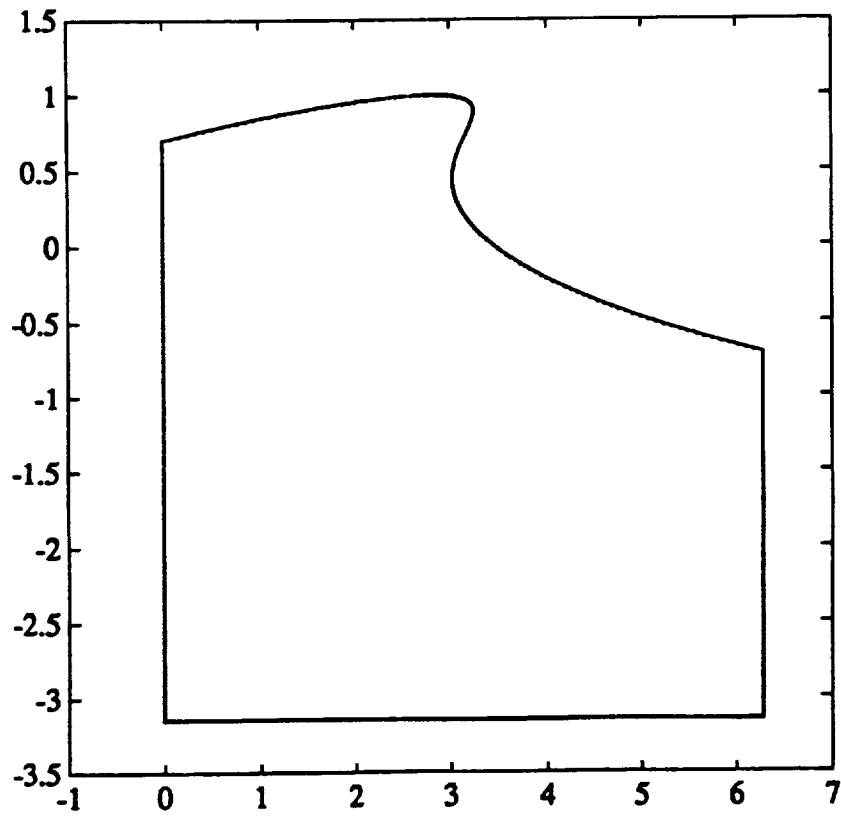


Figure 4. The configuration for Example 3.

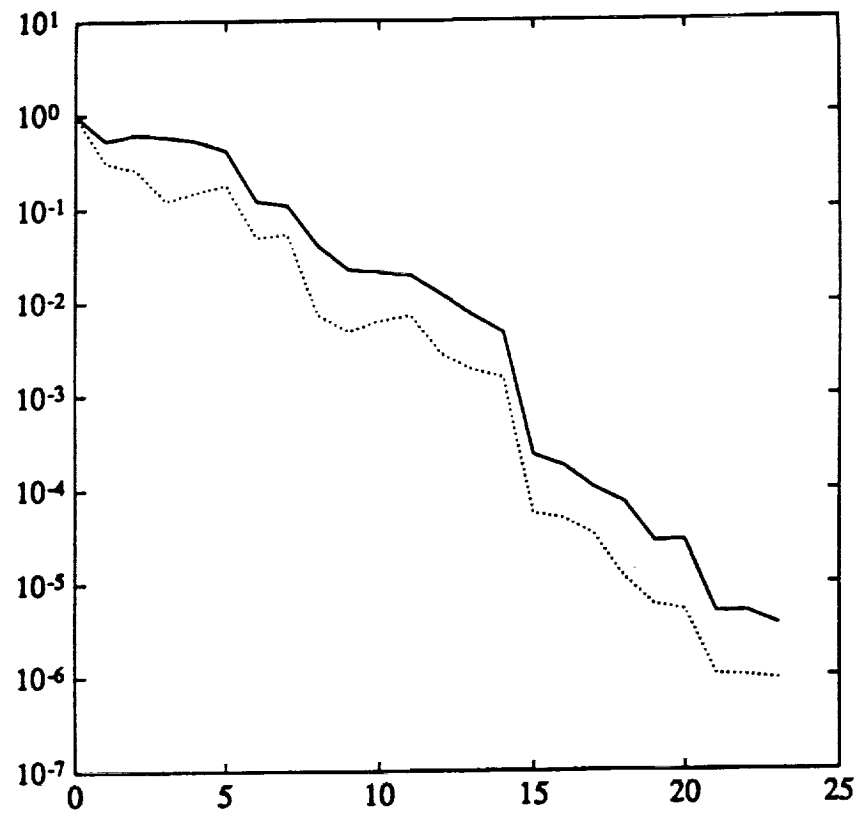


Figure 5. QMR convergence curves for Example 3, right SSOR.

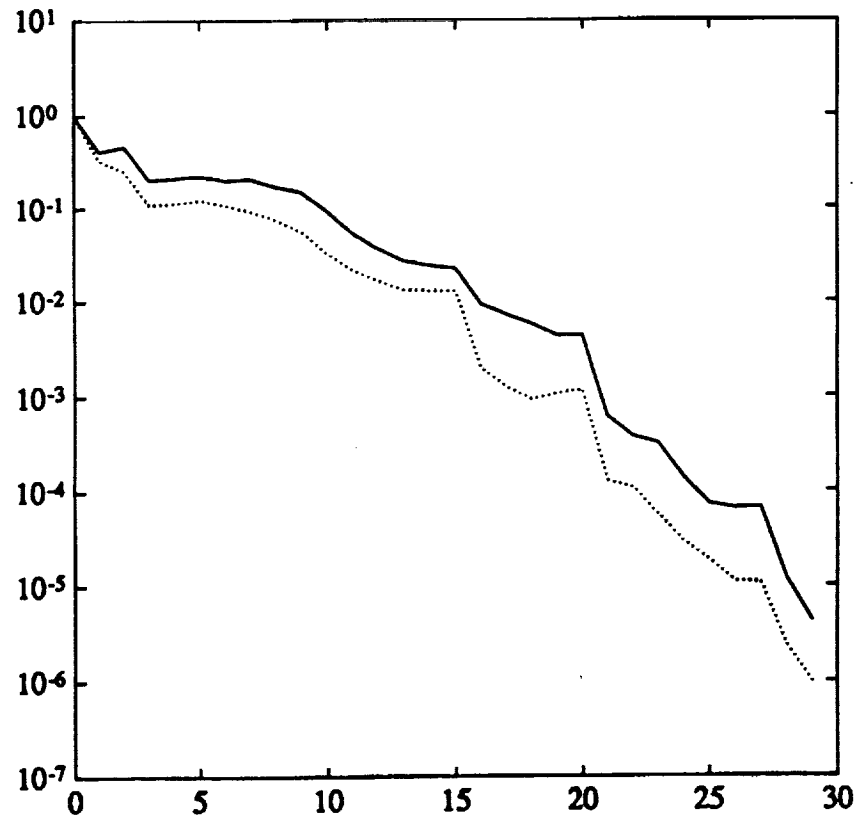


Figure 6. QMR convergence curves for Example 4, right SSOR.

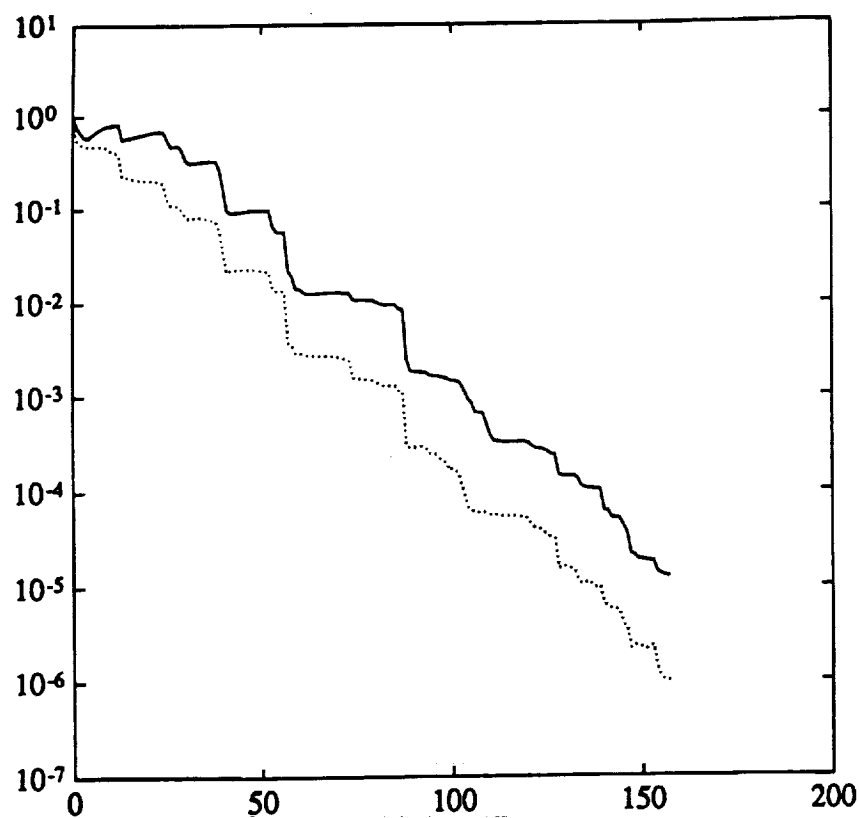


Figure 7. QMR convergence curves for Example 5, right SSOR.

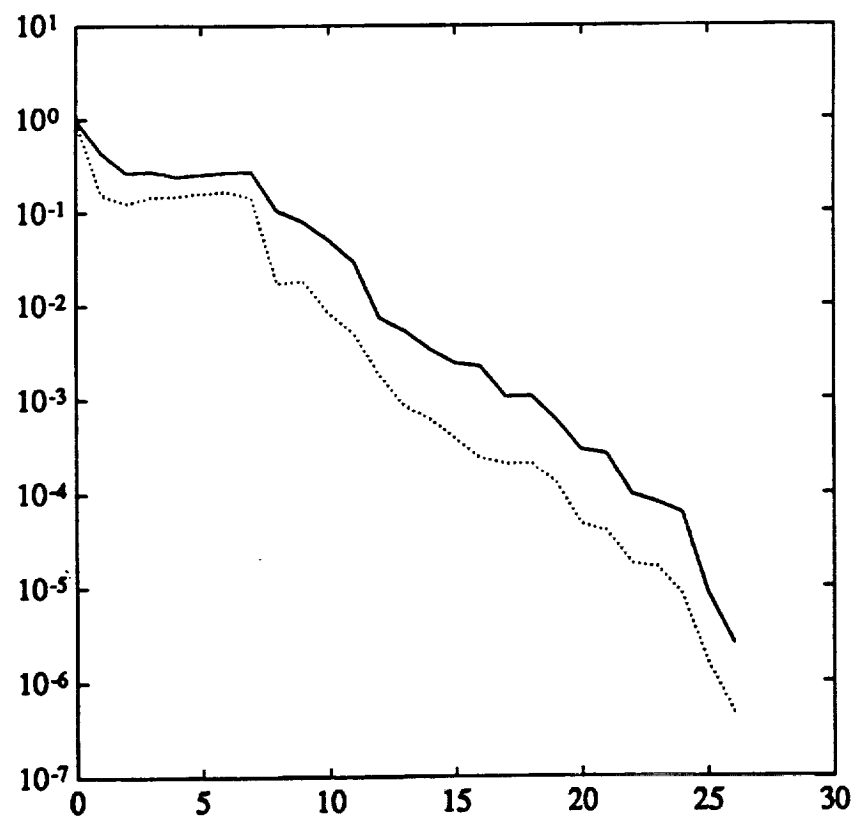


Figure 8. QMR convergence curves for Example 5, ILUT(0).

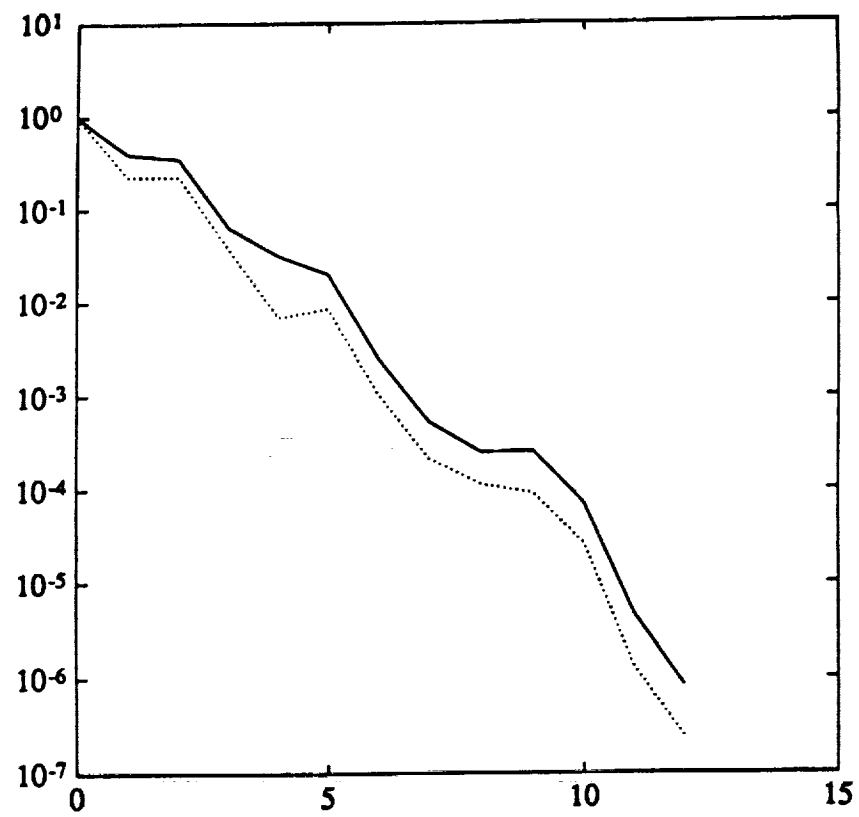


Figure 9. QMR convergence curves for Example 5, ILUT(4).

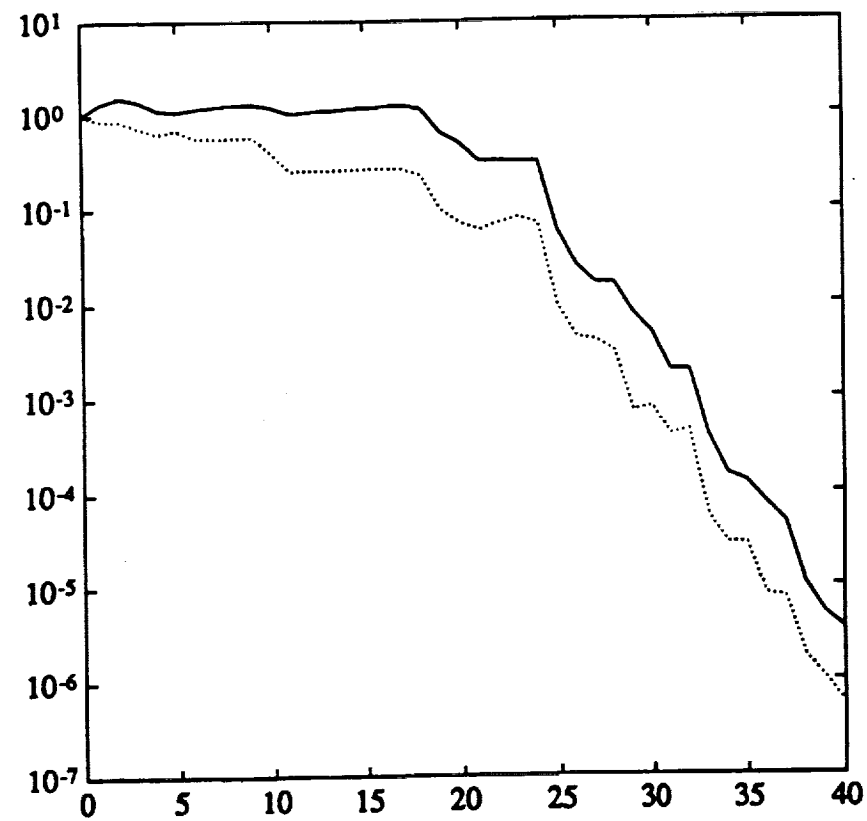


Figure 10. QMR convergence curves for Example 6, right SSOR.

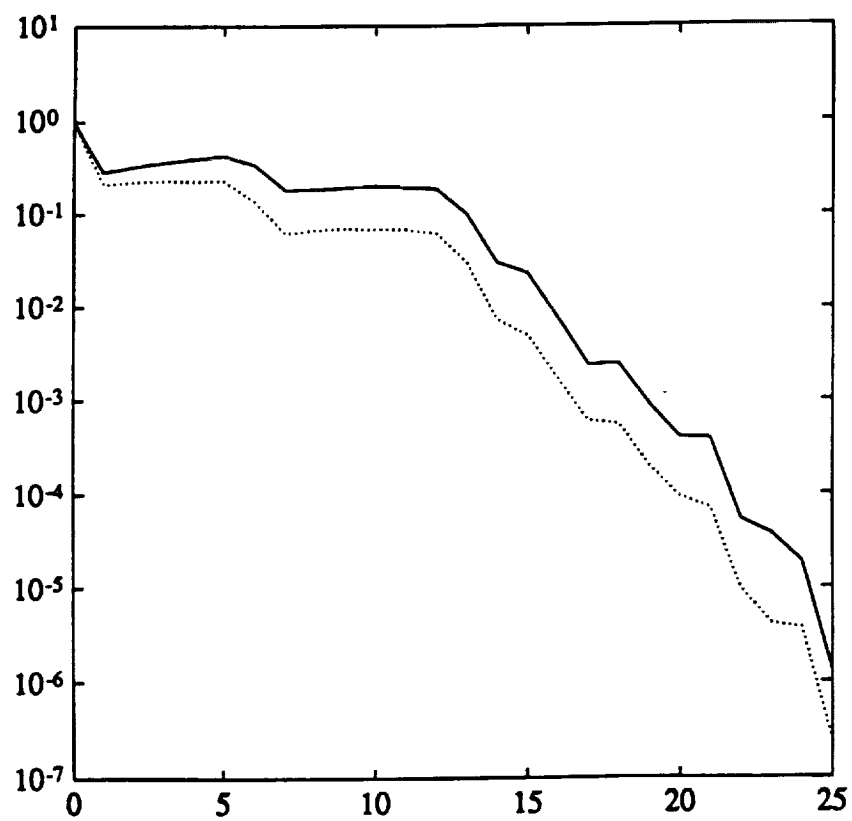


Figure 11. QMR convergence curves for Example 6. ILUT(0).

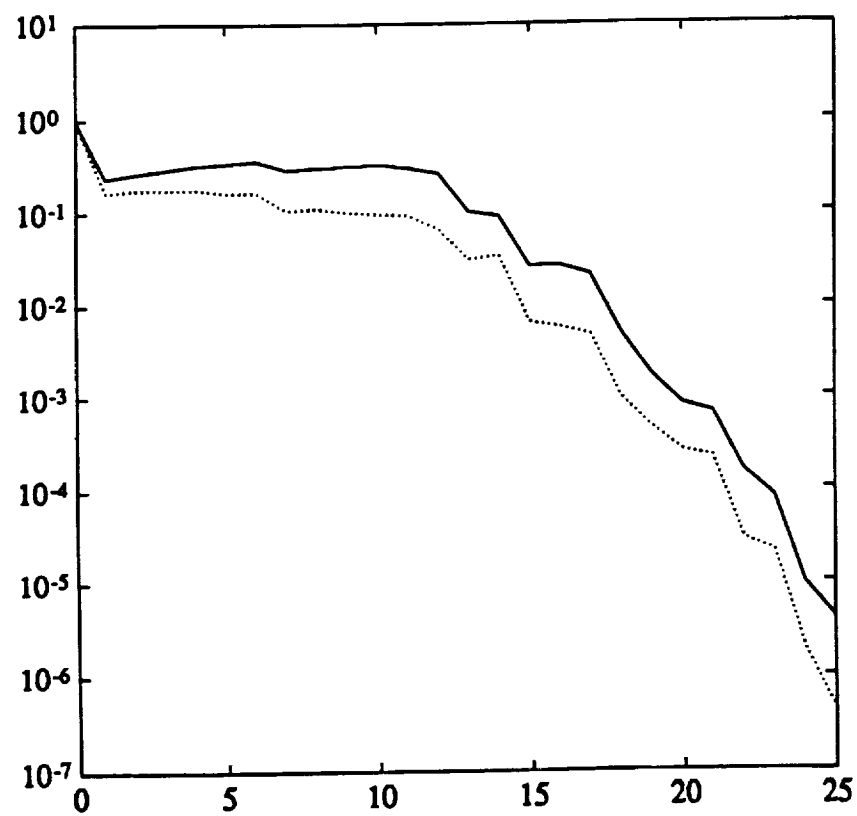


Figure 12. QMR convergence curves for Example 6, ILUT(4).

14. CONCLUDING REMARKS

We have proposed a robust iterative solver for non-Hermitian linear systems. Based on the look-ahead Lanczos algorithm described in Part I [6] of the paper, the method generates iterates which are characterized by a quasi-minimal residual (QMR) property. The QMR approach is closely related to the biconjugate gradient (BCG) algorithm; however, unlike BCG, the QMR algorithm has smooth convergence curves and good numerical properties.

For the case of real nonsymmetric matrices A , we have FORTRAN implementations of the QMR method and the underlying look-ahead Lanczos algorithm. These programs are listed in the Appendix. These codes are available electronically from the authors (na.freund@na-net.stanford.edu or na.nachtigal@na-net.stanford.edu).

Acknowledgements. The authors would like to thank Youcef Saad, who provided us with routines for generating test Example 2, and Dick Yue and Hongbo Xu for generating the matrices for test Example 3.

REFERENCES

- [1] BOLEY, D., ELHAY, S., GOLUB, G. H., AND GUTKNECHT, M. H. Nonsymmetric Lanczos and finding orthogonal polynomials associated with indefinite weights. Numerical Analysis Report NA-90-09, Stanford, August 1990.
- [2] CULLUM, J., AND WILLOUGHBY, R. A. A practical procedure for computing eigenvalues of large sparse nonsymmetric matrices. In *Large Scale Eigenvalue Problems*, J. Cullum and R.A. Willoughby, Eds., North-Holland, 1986, 193-240.
- [3] DOI, S., AND LICHNEWSKY, A. Some parallel and vector implementations of preconditioned iterative methods on Cray-2. *International Journal of High Speed Computing* 2 (1990), 143-179.
- [4] DUFF, I. S., GRIMES, R. G., AND LEWIS, J. G. Sparse matrix test problems. *ACM Trans. Math. Softw.* 15 (1989), 1-14.
- [5] FREUND, R. W. Conjugate gradient type methods for linear systems with complex symmetric coefficient matrices. Technical Report 89.54, RIACS, NASA Ames Research Center, December 1989.
- [6] FREUND, R. W., GUTKNECHT, M. H., AND NACHTIGAL, N. M. An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices, Part I. Technical Report 90.45, RIACS, NASA Ames Research Center, November 1990.
- [7] FREUND, R. W., AND NACHTIGAL, N. M. QMR: a quasi-minimal residual method for non-Hermitian linear systems. Technical Report, RIACS, NASA Ames Research Center, 1990, in preparation.
- [8] GOLUB, G. H., AND VAN LOAN, C. F. *Matrix computations*. First edition, The Johns Hopkins University Press, Baltimore, 1983.
- [9] LANCZOS, C. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand.* 49 (1952), 33-53.
- [10] MEIJERINK, J. A., AND VAN DER VORST, H. A. An iterative solution for linear systems of which the coefficient matrix is a symmetric M -matrix. *Math. Comp.* 31 (1977), 148-162.
- [11] PAIGE, C. C., AND SAUNDERS, M. A. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.* 12 (1975), 617-629.
- [12] PARLETT, B. N., AND SAAD, Y. Complex shift and invert strategies for real matrices. *Linear Algebra Appl.* 88/89 (1987), 575-595.
- [13] SAAD, Y. The Lanczos biorthogonalization algorithm and other oblique projection methods for solving large unsymmetric systems. *SIAM J. Numer. Anal.* 19 (1982), 485-506.

- [14] SAAD, Y. SPARSKIT: a basic tool kit for sparse matrix computations. Technical Report 90.20, RIACS, NASA Ames Research Center, May 1990.

APPENDIX

In this appendix we present FORTRAN codes listings for the Lanczos and QMR routines. We also make the correspondence between variables in the code and their name in this paper. Under each routine, each variable also appearing in the paper is listed as it appears in the code, followed by its name in the paper and an equation or a section number. The equation number listed is either the defining equation, if one exists, or the first equation where the variable appears, if a defining equation does not exist. In the latter case, the variable is usually defined in the text before or after the equation listed. The section number appears for variables which are defined in the introduction or in the first part of the notation section.

The work estimates for one step of this implementation of the look-ahead Lanczos algorithm are as follows:

- On the average, the algorithm builds 1×1 blocks, and requires:

1	multiplication by A
1	multiplication by A^T
$2 \cdot (n - n_{k-1} + 1)$	DAXPY operations of length N
4	DDOT operations of length N

- If building a pair of inner vectors, then the worst case requires:

1	multiplication by A
1	multiplication by A^T
$2 \cdot (n - n_{k-1} + 1) + 4$	DAXPY operations of length N
5	DDOT operations of length N
1	SVD of a $(n - n_{k-1} + 1) \times (n - n_{k-1} + 1)$ matrix
1	$(n - n_{k-1} + 1)^3$ work to compute the inverse

The additional work comes from the inner recursion (DAXPY), the SVD of a matrix of size bigger than 1 (and computing its inverse), and finally, the one dot product we have to recompute in cases when we first build a regular vector and then discover that it fails (4.12). In addition, when the Lanczos vectors are scaled, the algorithm performs two DSCAL operations of length N (we have ignored in the above $\mathcal{O}(n - n_{k-1} + 1)$ work).

The routines are organized as follows:

- DLAL

This routine is the lowest level routine, implementing one step of the Lanczos algorithm with look-ahead. The routine is called both by the eigenvalue code and by the linear systems' solver.

$$N = n \quad (3)$$

$$NK = n_k \quad (3)$$

$$NKM1 = n_{k-1} \quad (3)$$

$$NLEN = N \quad (1)$$

$$SC(1, i) = \frac{s_i}{s_{i-1}} \quad (3.1)$$

$$SC(2, i) = \frac{t_i}{t_{i-1}} \quad (3.1)$$

$$SC(3, i) = \frac{s_i}{t_i} \quad (3.1)$$

$$SC(4, i) = \xi_i \quad (3.2)$$

$$SC(5, i) = \sigma_i \quad (3.2)$$

$$TETA = \eta_n \quad (3.8)$$

$$TZETA = \zeta_n \quad (3.8)$$

$$VW(:, Q(i)) = \dot{v}_i \quad (3.2)$$

$$VW(:, M + Q(i)) = \dot{w}_i \quad (3.2)$$

$$WK(:, 1 : M) = (\hat{W}_k^T \hat{V}_k) \quad (3.7)$$

$$WK(:, M + 1 : 2 * M) = (\hat{W}_k^T \hat{V}_k)^{-1} \quad (3.7)$$

$$WK(:, 3 * M + 5) = (\hat{W}_{k-1}^T \hat{V}_{k-1})^{-1} \hat{W}_{k-1}^T \quad (4.1a)$$

$$WK(:, 3 * M + 6) = (\hat{W}_k^T \hat{V}_k)^{-1} \hat{W}_k^T \quad (4.1b)$$

- EIGLAL

This routine is the shell routine used in eigenvalue problems. It sets up the Hessenberg matrix that can be used to compute eigenvalue estimates for A, but does not actually compute the eigenvalue estimates. In general, some further processing is needed to compute the eigenvalues, and then identify and discard spurious and ghost eigenvalues.

$$H = \hat{H}^{(n)} \quad (3)$$

$$N = n \quad (3)$$

$$NK = n_k \quad (3)$$

$$\text{NKM1} = n_{k-1} \quad (3)$$

$$\text{NLEN} = N \quad (1)$$

$$\text{SC}(1, i) = \frac{s_i}{s_{i-1}} \quad (3.1)$$

$$\text{SC}(2, i) = \frac{t_i}{t_{i-1}} \quad (3.1)$$

$$\text{SC}(3, i) = \frac{s_i}{t_i} \quad (3.1)$$

$$\text{SC}(4, i) = \xi_i \quad (3.2)$$

$$\text{SC}(5, i) = \sigma_i \quad (3.2)$$

$$\text{VW}(:, \text{Q}(i)) = \hat{v}_i \quad (3.2)$$

$$\text{VW}(:, M + \text{Q}(i)) = \hat{w}_i \quad (3.2)$$

$$\text{WK}(:, 1 : M) = (\hat{W}_k^T \hat{V}_k) \quad (3.7)$$

$$\text{WK}(:, M + 1 : 2 * M) = (\hat{W}_k^T \hat{V}_k)^{-1} \quad (3.7)$$

$$\text{WK}(:, 3 * M + 5) = (\hat{W}_{k-1}^T \hat{V}_{k-1})^{-1} \hat{W}_{k-1}^T \quad (4.1a)$$

$$\text{WK}(:, 3 * M + 6) = (\hat{W}_k^T \hat{V}_k)^{-1} \hat{W}_k^T \quad (4.1b)$$

- SYSLAL

This routine is the shell routine used in solving linear system. It attempts to solve a linear system using the Lanczos-QMR algorithm.

$$\text{NLEN} = N \quad (1)$$

$$\text{SC}(1, i) = \frac{s_i}{s_{i-1}} = \rho_i \quad (3.1), (8.9)$$

$$\text{SC}(2, i) = \frac{t_i}{t_{i-1}} \quad (3.1)$$

$$\text{SC}(3, i) = \frac{s_i}{t_i} \quad (3.1)$$

$$\text{SC}(4, i) = \xi_i \quad (3.2)$$

$$\text{SC}(5, i) = \sigma_i \quad (3.2)$$

$$\text{VW}(:, \text{Q}(i)) = \hat{v}_i \quad (3.2)$$

$$\text{VW}(:, M + \text{Q}(i)) = \hat{w}_i \quad (3.2)$$

$$\text{VW}(:, 2 * M + 3) = b \quad (9.1)$$

$$\text{VW}(:, 2 * M + 4) = x_n \quad (10.14)$$

$$\text{VW}(:, 2 * M + 5 : 3 * M + 4) = [P_{k-1} \quad P_k] \quad (10.13)$$

$$\text{WK}(:, 1 : M) = (\hat{W}_k^T \hat{V}_k) \quad (3.7)$$

$$\text{WK}(:, M + 1 : 2 * M) = (\hat{W}_k^T \hat{V}_k)^{-1} \quad (3.7)$$

$$\text{WK}(:, 3 * M + 5) = (\hat{W}_{k-1}^T \hat{V}_{k-1})^{-1} \hat{W}_{k-1}^T \quad (4.1a)$$

$$\mathbf{WK}(:, 3 * \mathbf{M} + 6) = (\hat{\mathbf{W}}_k^T \hat{\mathbf{V}}_k)^{-1} \hat{\mathbf{W}}_k^T \quad (4.1b)$$

$$\mathbf{WK}(\mathbf{Q}(i), 3 * \mathbf{M} + 9) = \omega_i \quad (9.7)$$

$$\mathbf{WK}(\mathbf{Q}(i), 3 * \mathbf{M} + 10) = c_i \quad (10.2)$$

$$\mathbf{WK}(\mathbf{Q}(i), 3 * \mathbf{M} + 11) = s_i \quad (10.2)$$

$$\mathbf{WK}(\mathbf{Q}(i), 3 * \mathbf{M} + 12) = (t^{(n)})_i \quad (9.11)$$

$$\mathbf{WK}(\mathbf{Q}(i), 3 * \mathbf{M} + 13) = (Z_k^T)_i \quad (10.15)$$

$$\mathbf{WK}(:, 3 * \mathbf{M} + 15 : 4 * \mathbf{M} + 14) = Y_k \quad (10.15)$$

$$\mathbf{WK}(:, 4 : \mathbf{M} + 15 : 5 * \mathbf{M} + 14) = R_k \quad (10.15)$$

- DETA

This routine computes one of the coefficients for the inner recursion.

$$\mathbf{DETA}(i) = \eta_i \quad (3.8)$$

- DZETA

This routine computes one of the coefficients for the inner recursion.

$$\mathbf{DZETA}(i) = \zeta_i \quad (3.8)$$

- GETOMG

This routine computes the scaling factors ω_i for the QMR algorithm.

$$\mathbf{GETOMG}(i) = \omega_i \quad (9.7)$$

Copyright (C) 1990, Roland W. Freund and Noel M. Nachtigal
All rights reserved.

No part of this code may be reproduced, stored in a retrieval system, translated, transcribed, transmitted or distributed in any form or by any means means, manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, photocopying, recording, or otherwise, without the prior explicit written permission of the author(s) or their designated proxies. In no event shall the above copyright notice be removed or altered in any way.

This code is provided "as is", without any warranty of any kind, either expressed or implied, including but not limited to, any implied warranty of merchantability or fitness for any purpose. In no event will any party who distributed the code be liable for damages or for any claim(s) by any other party, including but not limited to, any lost profits, lost monies, lost data or data rendered inaccurate, losses sustained by third parties, or any other special, incidental or consequential damages arising out of the use or inability to use the program, even if the possibility of such damages has been advised against. The entire risk as to the quality, the performance, and the fitness of the program for any particular purpose lies with the party using the code.

ANY USE OF THIS CODE CONSTITUES ACCEPTANCE OF THE TERMS OF THE
ABOVE STATEMENTS

This file contains the basic routines for the look-ahead Lanczos algorithm. DLAL carries out one step of the algorithm and DSCALE is used to scale the Lanczos vectors. DEPS is used by DSCALE to compute machine epsilon, DADD is called by DEPS to ensure that no unwanted optimization takes place, and DZERO is called by DLAL to zero out vectors (it is also used elsewhere in the code).

SUBROUTINE DLAL (NDIM,NLEN,M,N,NK,NKM1,VW,SC,WK,Q,NORMS,TOL,TF,
\$ INFO)

Purpose:

This subroutine carries out one step of the look-ahead Lanczos algorithm. The matrix A is referenced solely through the external subroutines AXB and ATXB, which are of the form:

SUBROUTINE AXB (X,B) - computes $B = A * X$.
SUBROUTINE ATXB (X,B) - computes $B = A^T * X$.

Most of the inputs to this routine are not checked for validity; it is the responsibility of the caller to ensure that the various dimensions, indices, and data are valid, and that the output unit TF (when applicable) is ready for output. The only inputs checked are the tolerances in TOL, which, when they are non-positive, are replaced with default values as follows:

TOL(1) = 1/TOL(2)
TOL(2) = $\text{eps}^{\{1/2\}}$
TOL(3) = $\text{eps}^{\{1/4\}}$
TOL(4) = $\text{eps}^{\{1/3\}}$

Here, eps is machine epsilon.

Normally, this routine is not called directly from the top-most

level, but rather from an intermediate routine that uses Lanczos to solve either eigenvalue problems or linear systems. The caller initializes the following:

VW(:,Q(1)) = the first Lanczos vector V_1
 VW(:,M+Q(1)) = the first Lanczos vector W_1
 SC(:,1) = the scaling factors for V_1 and W_1
 WK(1,1) = the dot product $W_1^T V_1$
 Q = the array of wrapped indices
 NORMS(1) = the initial estimate for the norm of A
 TOL(1:4) = tolerances (optional)

Thereafter, it is usually left up to this routine to update the variables used. Upon exit, N, NK, NKM1, VW, SC, WK, NORMS, TOL, and INFO might be changed.

Parameters:

NDIM = the dimensioned size of the array VW (input).
 NLEN = the actual size of the Lanczos vectors V and W; this also implicitly determines the size of the matrix A (input).
 M = the maximum number of Lanczos vectors that can be stored in the array VW. It is related to the size of the largest block that can be built. The algorithm runs out of memory when the number of vectors in the last block and in the current block reaches M. For this reason, M must be at least 3; note that this is not checked (input).
 N = the index of the last pair of vectors (input/output).
 NK = the index of the last regular vectors (input/output).
 NKM1 = the index of the next-to-the-last regular vectors (input/output).
 VW = work array dimensioned (NDIM,2*M+2) words. It is used to store the Lanczos vectors V in VW(:,1:M), the vectors W in VW(:,M+1:2*M), and two temporary vectors used by DLAL in VW(:,2*M+1) and VW(:,2*M+2). The Lanczos vectors V and W are stored wrapped, i.e., V_N is stored in VW(:,Q(N)), and W_N is stored in VW(:,M+Q(N)), where Q(N) is assumed to be a wrapped index array -- see below (input/output).
 SC = work array dimensioned (5,M) words, used for the various scale factors. We have:
 SC(1,i) = S(i) / S(i-1)
 SC(2,i) = T(i) / T(i-1)
 SC(3,i) = S(i) / T(i)
 SC(4,i) = CSI(i)
 SC(5,i) = SIG(i)

Note that the scale routine DSCALE expects to receive the scale factors in a 5x1 vector as the one described above (input/output).

WK = work array dimensioned (M,3*M+6) words, used for internal variables. We have:

WK(:,1:M) = $(W_{\{NK\}}^T V_{\{NK\}})$
 WK(:,M+1:2*M) = $(W_{\{NK\}}^T V_{\{NK\}})^{-1}$
 WK(:,2*M+1:3*M) = work array for the SVD
 WK(:,3*M+1) = temporary vector
 WK(:,3*M+2) = temporary vector
 WK(:,3*M+3) = temporary vector
 WK(:,3*M+4) = the saved last column of the matrix $(W_{\{NKM1\}}^T V_{\{NKM1\}})^{-1}$
 WK(:,3*M+5) = H(NK:NKP1,N)
 WK(:,3*M+6) = H(NK:NKP1,N+1)

Of particular interest to the caller are H(NK:NKP1,N) and H(NK:NKP1,N+1), since they are parts of the columns of H. Usually, the caller will extract these after each call and either store them so as to form H (for eigenvalues) or use them to update X_N (for linear systems) (input/output).

Q = integer array specifying the indices for all the wrapped variables (V,W,SC,WK). To allow the algorithm to run more than M steps, these variable wrap around, in that Q(I) is

the index of the slots where that variables are stored at the I-th step. Normally, these indices would be in order, basically $Q(I) = I \text{ MOD } M + 1$, but the algorithm makes no assumptions to this effect. These indices are not checked in any way for validity (input).

NORMS = vector with estimates for the norm of A. NORMS(1) is the current estimate. It should be at least slightly larger than 1.0, to avoid the possible effects of roundoff for the identity matrix. For this reason, the routine will change it to 2.0 if it is smaller than 2.0. NORMS(2) is what the estimate should be to allow closure of blocks caused by the norm check. A value of 0.0 indicates that no estimates are available, i.e., all inner vectors were built due to the moment matrix being singular. The second estimate is usually used in conjunction with restarting a block if the algorithm runs out of memory (input/output).

TOL = vector with the tolerances used in the various checks. We have:

- TOL(1) = upper bound for the range of CSI and SIG
- TOL(2) = lower bound for the range of CSI and SIG
- TOL(3) = convergence tolerance for the norms of the Lanczos vectors
- TOL(4) = level below which the singular values of $(W_{\{NK\}}^T V_{\{NK\}})$ are considered zero

Note that the scale routine DSCALE expects to receive the first three tolerances in a 3x1 vector as described. The values supplied by the caller are checked for validity and default values (see above) are supplied if the user provides a non-positive value for any of the tolerances (input/output).

TF = output unit for a trace file. If TF non-zero, the routine will output to unit TF trace messages detailing execution decisions. The output unit is assumed to be available and ready (input).

INFO = information passing variable.

On input:

- INFO = 0 ==> proceed normally
- INFO = 1 ==> closing the block strongly recommended; if the block doesn't close naturally, do not update the counters, but update the norm estimate, if possible.

Upon exit:

- INFO = 0 ==> nothing to report
- INFO < 0 ==> the SVD routine returned this error code (but with positive sign)
- INFO = 1 ==> an A-invariant subspace has been found
- INFO = 2 ==> an A^T-invariant subspace has been found
- INFO = 3 ==> both subspaces have been found
- INFO = 4 ==> the block did not close, though strongly recommended (INFO=1 on input); updated norm estimate, but did not compute any vectors and did not update the counters.

(input/output).

External routines used:

subroutine axb(x,b)
 Computes $b = A * x$.

subroutine atxb(x,b)
 Computes $b = A^T * x$.

subroutine daxpy(n,da,dx,incx,dy,incy)
 Computes $dy = da * dx + dy$.

subroutine dcopy(n,dx,incx,dy,incy)
 Computes $dy = dx$.

double precision ddot(n,dx,incx,dy,incy)
 Computes $dy' * dx$.

double precision deta(i)

```

C      Computes the second recursion scalar for the inner vectors.
C      subroutine dscal(n,da,dx,incx)
C      Computes dx = da * dx.
C      subroutine dscale(nact,v,w,sc,tol)
C      Computes the scaling factors for v and w.
C      subroutine dsvdc(x,ldx,n,p,s,e,u,ldu,v,ldv,work,job,info)
C      Computes the singular value decomposition of x.
C      subroutine dzero(n,dx,incx)
C      Zeros out dx.
C      double precision dzeta(i)
C      Computes the first recursion scalar for the inner vectors.
C
C      Noel M. Nachtigal
C      August 28, 1990
C
C*****
C
C      INTRINSIC ABS, MAX, MIN
C      EXTERNAL DDOT, DEPS, DETA, DZETA
C      DOUBLE PRECISION DDOT, DEPS, DETA, DZETA
C
C      INTEGER INFO, M, N, NDIM, NK, NKM1, NLEN, Q(*), TF
C      DOUBLE PRECISION NORMS(2), SC(5,M), TOL(4)
C      DOUBLE PRECISION VW(NDIM,2*M+2), WK(M,3*M+6)
C
C      Local variables.
C
C      INTEGER HBASE, I, J, LCL, LINFO, NP1
C      DOUBLE PRECISION ANORM, INVCSI, INVSIG, DTMP, DTMP1, DTMP2
C      DOUBLE PRECISION NUNORM, TETA, TZETA, WNV, WNP1V
C      LOGICAL INNER
C
C      Check the tolerances and the norm estimate.
C
C      IF (TOL(2).LE.0.0) TOL(2) = DEPS()**(1.0/2.0)
C      IF (TOL(1).LE.0.0) TOL(1) = 1.0 / DEPS()
C      IF (TOL(3).LE.0.0) TOL(3) = DEPS()**(1.0/4.0)
C      IF (TOL(4).LE.0.0) TOL(4) = DEPS()**(1.0/3.0)
C      NORMS(1) = MAX(NORMS(1),2.0D0)
C
C      Compute local counters.
C
C      NP1    = N + 1
C      LCL    = N - NK + 1
C      HBASE  = 1 - NKM1
C      IF (NKM1.EQ.0) HBASE = 0
C
C      Initialize the norm estimate.
C
C      ANORM  = NORMS(1)
C      NUNORM = NORMS(2)
C
C      Zero out the work portion of the column of H.
C
C      CALL DZERO (LCL,WK(HBASE+NK,3*M+5),1)
C
C      Compute the matrix-vector products.
C
C      CALL AXB (VW(1,Q(N)),VW(1,2*M+1))
C      CALL ATXB (VW(1,M+Q(N)),VW(1,2*M+2))
C
C      Subtract the part common to both types of vectors. This is also
C      done to enhance the numerical properties.
C
C      INVSIG = 1.0 / SC(5,Q(N))
C      INVCSI = 1.0 / SC(4,Q(N))

```

```

      IF (NKM1.GT.0) THEN
        DO 10 I = NKM1, NK-1
          DTMP2 = WK(HBASE+I,3*M+5)
          DTMP1 = INVSIG * SC(5,Q(I)) * DTMP2
          CALL DAXPY (NLEN,-DTMP1,VW(1,Q(I)),1,VW(1,2*M+1),1)
          DTMP1 = INVCSI * SC(4,Q(I)) * DTMP2 * SC(3,Q(N))
$          / SC(3,Q(I))
          CALL DAXPY (NLEN,-DTMP1,VW(1,M+Q(I)),1,VW(1,2*M+2),1)
10      CONTINUE
      END IF

C
C      Compute the inner product (W_N^T A \tilde{V}_N).
C
      WNV = DDOT(NLEN,VW(1,M+Q(N)),1,VW(1,2*M+1),1)

C
C      Compute the SVD of (W_{NK}^T V_{NK}). We have to copy it first to
C      a temporary array, since the DSVDC routine destroys its argument.
C      From the DSVDC routine, we want both singular values and singular
C      vectors, hence JOB = 11.
C
      DO 20 J = 1, LCL
        CALL DCOPY (LCL,WK(1,J),1,WK(1,M+J),1)
20      CONTINUE
      CALL DSVDC (WK(1,M+1),M,LCL,LCL,WK(1,3*M+3),WK(1,3*M+2),
$      WK(1,2*M+1),M,WK(1,M+1),M,WK(1,3*M+1),11,LINFO)

C
C      Check for error in DSVDC; abort on error.
C
      LINFO = -LINFO
      IF (LINFO.NE.0) GO TO 120

C
C      Check the smallest singular value to determine singularity of the
C      moment matrix (W_{NK}^T V_{NK}).
C
      DTMP = WK(1,3*M+3)
      DO 30 I = 2, LCL
        DTMP = MIN(DTMP,WK(I,3*M+3))
30      CONTINUE
      INNER = DTMP.LE.TOL(4)
      IF (INNER.AND.(TF.NE.0)) THEN
        WRITE (TF,'(A7,I5,A21)') 'Vector ',NP1,' is an inner vector; '
        WRITE (TF,'(A31,E10.3)') '==> moment matrix is singular: ',DTMP
      END IF

C
C      IF (.NOT.INNER) THEN
C
C      Matrix is not singular, compute its inverse. WK(:,M+I) has the
C      right singular vectors, WK(:,2*M+I) has the left singular vectors
C      and WK(:,3*M+3) contains the singular values. We save the inverse
C      in WK(:,M+I).
C
      DO 40 I = 1, LCL
        CALL DSCAL (LCL,1.0/WK(I,3*M+3),WK(1,M+I),1)
40      CONTINUE
      DO 60 I = 1, LCL
        DO 50 J = 1, LCL
          WK(J,3*M+2) = DDOT(LCL,WK(I,M+1),M,WK(J,2*M+1),M)
50          CONTINUE
        CALL DCOPY (LCL,WK(1,3*M+2),1,WK(I,M+1),M)
60      CONTINUE

C
C      Compute the (W_{NK}^T A V_N) term.
C
      WK(LCL,3*M+2) = SC(5,Q(N)) * SC(4,Q(N)) * WNV
      DO 70 I = 1, LCL-1
        DTMP = SC(2,Q(NK+I)) * WK(I+1,LCL) + DZETA(I-1) * WK(I,LCL)

```

```

        IF (I.GT.1) THEN
            DTMP = DTMP + DETA(I-1) * WK(I-1,LCL) / SC(2,Q(NK+I-1))
        END IF
        WK(I,3*M+2) = DTMP
70    CONTINUE
C
C    Compute  $H(NK:N,N) = (W_{\{NK\}}^T V_{\{NK\}})^{-1} W_{\{NK\}}^T A V_N$ .
C    Check whether we can build a regular vector with it.
C
        DTMP1 = 0.0
        DTMP2 = 0.0
        DO 80 I = 1, LCL
            WK(I,3*M+1) = DDOT(LCL,WK(I,M+1),M,WK(1,3*M+2),1)
            DTMP = ABS(WK(I,3*M+1))
            DTMP1 = DTMP1 + DTMP
            DTMP2 = DTMP2 + DTMP / SC(3,Q(NK+I-1))
80    CONTINUE
        DTMP2 = SC(3,Q(N)) * DTMP2
        INNER = (DTMP1.GT.ANORM).OR.(DTMP2.GT.ANORM)
        IF (INNER) THEN
C
C    One or both of the norm checks has failed, we have to build an
C    inner vector. Output trace messages and update norm estimate.
C
            IF (TF.NE.0) THEN
                WRITE (6,'(A7,I5,A21)')
                'Vector ',NP1,' is an inner vector;'
                IF (DTMP1.GT.ANORM) WRITE (6,'(A29,E10.3)')
                '==> second term in V is bad: ',DTMP1/ANORM
                IF (DTMP2.GT.ANORM) WRITE (6,'(A29,E10.3)')
                '==> second term in W is bad: ',DTMP2/ANORM
            END IF
            DTMP1 = MAX(DTMP1,DTMP2)
            IF (NUNORM.EQ.0.0) THEN
                NUNORM = DTMP1
            ELSE
                NUNORM = MIN(NUNORM,DTMP1)
            END IF
        END IF
    END IF
C
    IF (.NOT.INNER) THEN
C
C    We can build a regular vector, let's do it. Copy the temporary
C    vectors to the proper slots in V and W.
C
        CALL DCOPY (NLEN,VW(1,2*M+1),1,VW(1,Q(NP1)),1)
        CALL DCOPY (NLEN,VW(1,2*M+2),1,VW(1,M+Q(NP1)),1)
C
C    Add in the term  $V_{\{NK\}} H(NK:N,N)$ .
C
        DO 90 I = NK, N
            DTMP2 = WK(I-NK+1,3*M+1)
            DTMP1 = INVSIG * SC(5,Q(I)) * DTMP2
            CALL DAXPY (NLEN,-DTMP1,VW(1,Q(I)),1,VW(1,Q(NP1)),1)
            DTMP1 = INVCSI * SC(4,Q(I)) * DTMP2 * SC(3,Q(N))
            / SC(3,Q(I))
            CALL DAXPY (NLEN,-DTMP1,VW(1,M+Q(I)),1,VW(1,M+Q(NP1)),1)
90    CONTINUE
C
C    Scale the new vectors.
C
        SC(3,Q(NP1)) = SC(3,Q(N))
        SC(4,Q(NP1)) = SC(4,Q(N))
        SC(5,Q(NP1)) = SC(5,Q(N))
        CALL DSCALE (NLEN,VW(1,Q(NP1)),VW(1,M+Q(NP1)),SC(1,Q(NP1)),TOL)

```

```

      WK(HBASE+NP1,3*M+5) = SC(1,Q(NP1))
C
C      Compute the inner product (W_{N+1}^T V_{N+1}).
C
      WNP1V = DDOT(NLEN,VW(1,M+Q(NP1)),1,VW(1,Q(NP1)),1)
C
C      Compute the term W_{NK}^T A V_{N+1} and its 1-norm.
C
      CALL DCOPY (LCL,WK(1,M+LCL),1,WK(1,3*M+2),1)
      DTMP = SC(5,Q(NP1)) * SC(4,Q(NP1)) * SC(2,Q(NP1)) * WNP1V
      CALL DSCAL (LCL,DTMP,WK(1,3*M+2),1)
C
C      Check whether we can build either vector at the next step.
C
      DTMP1 = 0.0
      DTMP2 = 0.0
      DO 100 I = 1, LCL
        DTMP = ABS(WK(I,3*M+2))
        DTMP1 = DTMP1 + DTMP
        DTMP2 = DTMP2 + DTMP / SC(3,Q(NK+I-1))
100    CONTINUE
      DTMP2 = SC(3,Q(N)) * DTMP2
      INNER = (DTMP1.GT.ANORM).OR.(DTMP2.GT.ANORM)
C
C      The next vector would be bad, we have to build an inner vector.
C      Output trace messages and update the norm estimate.
C
      IF (INNER) THEN
        IF (TF.NE.0) THEN
          WRITE (6,'(A7,I5,A21)')
$          'Vector ',NP1,' is an inner vector;'
          IF (DTMP1.GT.ANORM) WRITE (6,'(A30,E10.3)')
$          '==> next vector V will be bad:',DTMP1/ANORM
          IF (DTMP2.GT.ANORM) WRITE (6,'(A30,E10.3)')
$          '==> next vector W will be bad:',DTMP2/ANORM
          END IF
          DTMP1 = MAX(DTMP1,DTMP2)
          IF (NUNORM.EQ.0.0) THEN
            NUNORM = DTMP1
          ELSE
            NUNORM = MIN(NUNORM,DTMP1)
          END IF
        END IF
      END IF
C
      IF (INNER) THEN
C
C      Check whether we were supposed to close the block. If so, return
C      without computing anything else.
C
        IF (INFO.NE.0) THEN
          LINFO = 4
          GO TO 120
        END IF
C
C      We are building an inner vector. Copy the temporary vectors to
C      the proper slots in V and W.
C
        CALL DCOPY (NLEN,VW(1,2*M+1),1,VW(1,Q(NP1)),1)
        CALL DCOPY (NLEN,VW(1,2*M+2),1,VW(1,M+Q(NP1)),1)
C
C      Add the terms from the inner vector recursion.
C
      TZETA = DZETA(N-NK)
      WK(HBASE+N,3*M+5) = TZETA
      CALL DAXPY (NLEN,-TZETA,VW(1,Q(N)),1,VW(1,Q(NP1)),1)

```

```

CALL DAXPY (NLEN,-TZETA,VW(1,M+Q(N)),1,VW(1,M+Q(NP1)),1)
IF (LCL.GT.1) THEN
  TETA = DETA(N-NK)
  DTMP2 = TETA * INVSIG * SC(5,Q(N-1)) / SC(1,Q(N))
  CALL DAXPY (NLEN,-DTMP2,VW(1,Q(N-1)),1,VW(1,Q(NP1)),1)
  DTMP2 = TETA * INVCSI * SC(4,Q(N-1)) / SC(2,Q(N))
  CALL DAXPY (NLEN,-DTMP2,VW(1,M+Q(N-1)),1,VW(1,M+Q(NP1)),1)
  WK(HBASE+N-1,3*M+5) = TETA / SC(1,Q(N))
END IF

C
C Scale the new vectors.
C
  SC(3,Q(NP1)) = SC(3,Q(N))
  SC(5,Q(NP1)) = SC(5,Q(N))
  SC(4,Q(NP1)) = SC(4,Q(N))
  CALL DSCALE (NLEN,VW(1,Q(NP1)),VW(1,M+Q(NP1)),SC(1,Q(NP1)),TOL)
  WK(HBASE+NP1,3*M+5) = SC(1,Q(NP1))

C
C Compute the inner product (W_{N+1}^T V_{N+1}).
C
  WNP1V = DDOT(NLEN,VW(1,M+Q(NP1)),1,VW(1,Q(NP1)),1)

C
C Update the matrix (W_{NK}^T V_{NK}).
C
  WK(LCL+1,LCL+1) = SC(5,Q(NP1)) * SC(4,Q(NP1)) * WNP1V
  DTMP = SC(5,Q(N)) * SC(4,Q(N)) * WNV - TZETA * WK(LCL,LCL)
  IF (LCL.GT.1) THEN
    DTMP = DTMP - TZETA * WK(LCL,LCL-1) / SC(1,Q(N))
  END IF
  WK(LCL,LCL+1) = DTMP / SC(1,Q(NP1))
  WK(LCL+1,LCL) = DTMP / SC(2,Q(NP1))
  DTMP2 = SC(3,Q(NP1)) / SC(3,Q(N))
  DO 110 I = LCL-1, 1, -1
    DTMP1 = (DZETA(I) - TZETA) * WK(I,LCL)
    $      - TETA * WK(I,LCL-1) / SC(1,Q(N))
    $      + SC(2,Q(NK+I)) * WK(I+1,LCL)
    IF (I.GT.1) THEN
      DTMP1 = DTMP1 + DETA(I) * WK(I-1,LCL) / SC(2,Q(NK+I-1))
    END IF
    DTMP1 = DTMP1 / SC(1,Q(NP1))
    DTMP2 = DTMP2 * SC(3,Q(NK+I)) / SC(3,Q(NK+I-1))
    WK(LCL+1,I) = DTMP1 * DTMP2
    WK(I,LCL+1) = DTMP1
110  CONTINUE

C
C Initialize H(NKM1:NK-1,N+1) for the next step. It is the last
C column of (W_{NKM1}^T V_{NKM1})^{-1}, scaled appropriately.
C
  IF (NKM1.GT.0) THEN
    DTMP = SC(2,Q(NK)) * WK(1,LCL+1)
    CALL DCOPY (NK-NKM1,WK(1,3*M+4),1,WK(1,3*M+6),1)
    CALL DSCAL (NK-NKM1,DTMP,WK(1,3*M+6),1)
  END IF

C
ELSE

C
C We have built a regular vector. Output trace message.
C
  IF (TF.NE.0) WRITE (6,'(A7,I5,A21)')
  $      'Vector ',NP1,' is a regular vector.'

C
C Save H(NK:NKP1-1,N).
C
  CALL DCOPY (LCL,WK(1,3*M+1),1,WK(HBASE+NK,3*M+5),1)

C
C Save the last column of (W_{NK}^T V_{NK})^{-1} for next step.

```

```

C      CALL DCOPY (LCL,WK(1,M+LCL),1,WK(1,3*M+4),1)
C
C      Initialize H(NK:NKP1-1,N+1) for next step.
C
C      CALL DCOPY (LCL,WK(1,3*M+2),1,WK(1,3*M+6),1)
C      WK(1,1) = SC(5,Q(NP1)) * SC(4,Q(NP1)) * WNP1V
C
C      Update the counters.
C
C      NKM1 = NK
C      NK   = N + 1
C      END IF
C
C      Update the running counter.
C
C      N = N + 1
C
C      Check for termination.
C
C      LINFO = 0
C      IF (SC(4,Q(NP1)).EQ.-1.0) LINFO = LINFO + 1
C      IF (SC(5,Q(NP1)).EQ.-1.0) LINFO = LINFO + 2
C
C      Save the updated norm estimate and set the INFO variable.
C
120  NORMS(2) = NUNORM
      INFO = LINFO
C
C      RETURN
C      END
C
C*****
C
C      DOUBLE PRECISION FUNCTION DADD (X)
C
C      Purpose:
C      Computes X + 1.0. Used by the DEPS function to ensure that the
C      optimizer doesn't affect the results.
C
C      Parameters:
C      X = the variable to add 1.0 to (input).
C
C      Noel M. Nachtigal
C      November 18, 1987
C
C      DOUBLE PRECISION X
C
C      DADD = X + 1.0
C
C      RETURN
C      END
C
C*****
C
C      DOUBLE PRECISION FUNCTION DEPS ()
C
C      Purpose:
C      Computes double precision machine epsilon, the smallest number
C      which, when added to 1.0, gives 1.0. This function could use the
C      radix of the machine to obtain a better estimate, but its result
C      is good enough for our purposes. It does attempt to ensure that
C      any optimization does not affect the result.
C
C      External routines used:
C      double precision dadd(dx)

```

```

C      Computes dx = dx + 1. Used to get around optimizers.
C
C      Noel M. Nachtigal
C      October 25, 1990
C
C      EXTERNAL DADD
C      DOUBLE PRECISION DADD
C
C      Local variables.
C
C      DOUBLE PRECISION DTMP
C
C      DTMP = 1.0
C
C 30  IF (DADD(DTMP).GT.1.0) THEN
C      DTMP = DTMP / 2.0
C      GO TO 30
C  END IF
C
C      DEPS = DTMP
C
C      RETURN
C      END
C
C*****
C
C      SUBROUTINE DSCALE (N,V,W,SC,TOL)
C
C      Purpose:
C      Does scaling in the Lanczos algorithm. Scales the vectors V and W
C      to unit length. Also checks for invariant subspaces. Note that it
C      does not check its inputs for validity.
C
C      Parameters:
C      N      = the length of the vectors (input).
C      V      = the Lanczos vector V, scaled on output (input/output).
C      W      = the Lanczos vector W, scaled on output (input/output).
C      SC(1)  =  $S_{N+1} / S_N$  (output).
C      SC(2)  =  $T_{N+1} / T_N$  (output).
C      SC(3)  = on input,  $S_N / T_N$ ; on exit,  $S_{N+1} / T_{N+1}$  (input/
C              output).
C      SC(4)  = on input, CSI_N; on exit, CSI_{N+1}. If it is -1.0, then
C              the norm of W was less than TOL(3) on input, indicating
C              an invariant subspace for  $A^T$  (input/output).
C      SC(5)  = on input, SIGMA_N; on exit, SIGMA_{N+1}. If it is -1.0,
C              then the norm of V was less than TOL(3) on input,
C              indicating an invariant subspace for A (input/output).
C      TOL(1) = the level above which vectors will be scaled (input).
C      TOL(2) = the level below which vectors will be scaled (input).
C      TOL(3) = the tolerance level below which the norms of the vectors
C              are treated as zero (input).
C
C      External routines used:
C      double precision dnrm2(n,dx,incx)
C          Returns the 2-norm of dx.
C      subroutine dscal(n,da,dx,incx)
C          Computes dx = da * dx.
C
C      Noel M. Nachtigal
C      August 28, 1990
C
C      EXTERNAL DNRM2
C      DOUBLE PRECISION DNRM2
C
C      INTEGER N
C      DOUBLE PRECISION SC(*), TOL(3), V(*), W(*)

```

```

C
C      Local variables.
C
C      DOUBLE PRECISION TMPS,TMPT
C
C      Initialize the scale factors.
C
C      SC(1) = DNRM2(N,V,1)
C      SC(2) = DNRM2(N,W,1)
C      IF (SC(1)*SC(5).LE.TOL(3)) SC(5) = -1.0
C      IF (SC(2)*SC(4).LE.TOL(3)) SC(4) = -1.0
C      IF ((SC(4).LT.0.0).OR.(SC(5).LT.0.0)) RETURN
C
C      Compute the scale factors and scale the vectors.
C
C      TMPS = 1.0 / SC(1)
C      TMPT = 1.0 / SC(2)
C
C      IF ((TMPS.LE.TOL(2)).OR.(TMPS.GE.TOL(1))) THEN
C          CALL DSCAL (N,TMPS,V,1)
C          TMPS = 1.0
C      END IF
C      IF ((TMPT.LE.TOL(2)).OR.(TMPT.GE.TOL(1))) THEN
C          CALL DSCAL (N,TMPT,W,1)
C          TMPT = 1.0
C      END IF
C
C      SC(1) = SC(5) * SC(1)
C      SC(2) = SC(4) * SC(2)
C      SC(3) = SC(1) * SC(3) / SC(2)
C      SC(4) = TMPT
C      SC(5) = TMPS
C
C      RETURN
C      END
C
C*****
C
C      SUBROUTINE DZERO (N,DX,INCX)
C
C      Purpose:
C      Zeroes out DX.
C
C      Parameters:
C      N      = the length of the vector DX (input).
C      DX     = the vector to be zeroed out (output).
C      INCX   = the increment in the vector DX (input).
C
C      Noel M. Nachtigal
C      October 26, 1990
C
C      INTRINSIC MOD
C
C      INTEGER INCX, N
C      DOUBLE PRECISION DX(*)
C
C      Local variables.
C
C      INTEGER I, IX, M, MP1
C
C      IF (N.LE.0) RETURN
C      IF (INCX.NE.1) GO TO 40
C
C      Code for increment equal to 1.
C
C      M = MOD(N,8)

```

```

        IF (M.EQ.0) GO TO 20
        DO 10 I = 1, M
            DX(I) = 0.0
10      CONTINUE
        IF (N.LT.8) RETURN
20      MP1 = M + 1
        DO 30 I = MP1, N, 8
            DX(I) = 0.0
            DX(I+1) = 0.0
            DX(I+2) = 0.0
            DX(I+3) = 0.0
            DX(I+4) = 0.0
            DX(I+5) = 0.0
            DX(I+6) = 0.0
            DX(I+7) = 0.0
30      CONTINUE
        RETURN
C
C      Code for increment not equal to 1.
C
40      IX = 1
        IF (INCX.LT.0) IX = (-N+1) * INCX + 1
        DO 50 I = 1, N
            DX(IX) = 0.0
            IX = IX + INCX
50      CONTINUE
C
        RETURN
        END
C
C*****

```

Copyright (C) 1990, Roland W. Freund and Noel M. Nachtigal
All rights reserved.

No part of this code may be reproduced, stored in a retrieval system, translated, transcribed, transmitted or distributed in any form or by any means means, manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, photocopying, recording, or otherwise, without the prior explicit written permission of the author(s) or their designated proxies. In no event shall the above copyright notice be removed or altered in any way.

This code is provided "as is", without any warranty of any kind, either expressed or implied, including but not limited to, any implied warranty of merchantability or fitness for any purpose. In no event will any party who distributed the code be liable for damages or for any claim(s) by any other party, including but not limited to, any lost profits, lost monies, lost data or data rendered inaccurate, losses sustained by third parties, or any other special, incidental or consequential damages arising out of the use or inability to use the program, even if the possibility of such damages has been advised against. The entire risk as to the quality, the performance, and the fitness of the program for any particular purpose lies with the party using the code.

ANY USE OF THIS CODE CONSTITUTES ACCEPTANCE OF THE TERMS OF THE
ABOVE STATEMENTS

This file contains the routines for an eigenvalue solver which uses the Lanczos code. EIGLAL is the basic routine used to obtain the Hessenberg matrix whose eigenvalues are taken as estimates the eigenvalues of A.

SUBROUTINE EIGLAL (NDIM,NLEN,HDIM,NLIM,M,VW,SC,WK,Q,TOL,ANORM,
\$ INFO,H)

Purpose:

This subroutine uses the Lanczos algorithm to set up a Hessenberg matrix that can be used to compute eigenvalue estimates for A. It runs the Lanczos algorithm for NLIM steps, storing the columns of H returned by DLAL in H. The caller initializes the following:

VW(:,Q(1)) = the first Lanczos vector V₁
VW(:,M+Q(1)) = the first Lanczos vector W₁
Q = the array of wrapped indices
TOL(1:4) = tolerances for the Lanczos algorithm (optional)
ANORM = estimate for the norm of the matrix (optional)
INFO = the output units, if any

If the user provides non-positive values for the tolerances in TOL(1:4), the Lanczos routine DLAL will supply its own defaults. Also, if the user provides a norm estimate of less than 2.0, the Lanczos routine DLAL will set the norm estimate to 2.0.

Parameters:

NDIM = the dimensioned size of the array VW. Must be at least 1; checked for validity (input).

NLEN = the actual size of the Lanczos vectors V and W; this also implicitly determines the size of the matrix A. Must be less than or equal to NDIM; checked for validity (input).

```

C   HDIM  = the dimensioned size of H. Must be at least 1; checked
C           for validity (input).
C   NLIM  = the maximum number of steps the algorithm can take before
C           H overflows. Must be less than or equal to HDIM; checked
C           for validity. On exit, it is the size of the usable part
C           of H, i.e., the eigenvalues of  $H(1:NLIM,1:NLIM)$  can be
C           used as estimates for NLIM of the eigenvalues of A
C           (input/output).
C   M     = the maximum number of Lanczos vectors that can be stored
C           in the array VW. It is related to the size of the largest
C           block that can be built. The algorithm runs out of memory
C           when the number of vectors in the last block and in the
C           current block reaches M. For this reason, M must be at
C           least 3; checked for validity (input).
C   VW    = work array dimensioned (NDIM,2*M+2) words. It is used to
C           store the Lanczos vectors V in  $VW(:,1:M)$ , the vectors W
C           in  $VW(:,M+1:2*M)$ , and two temporary vectors used by DLAL
C           in  $VW(:,2*M+1)$  and  $VW(:,2*M+2)$ . The Lanczos vectors V and
C           W are stored wrapped, i.e.,  $V_N$  is stored in  $VW(:,Q(N))$ 
C           and  $W_N$  is stored in  $VW(:,M+Q(N))$ , where  $Q(N)$  is assumed
C           to be a wrapped index array -- see below (input/output).
C   SC    = work array dimensioned (5,M), used to store the various
C           scale factors. We have:
C           SC(1,i) = S(i) / S(i-1)
C           SC(2,i) = T(i) / T(i-1)
C           SC(3,i) = S(i) / T(i)
C           SC(4,i) = CSI(i)
C           SC(5,i) = SIG(i)
C           Note that the scale routine DSCALE expects to receive the
C           scale factors in a 5x1 vector as the one described above.
C           This routine initializes the first column; thereafter,
C           the DLAL routine will update the array (input/output).
C   WK    = work array dimensioned (M,5*M+14), used to store internal
C           variables. We have:
C           WK(:,1:M)      =  $(W_{\{NK\}}^T V_{\{NK\}})$ 
C           WK(:,M+1) to WK(:,2*M)
C                           =  $(W_{\{NK\}}^T V_{\{NK\}})^{-1}$ 
C           WK(:,2*M+1) to WK(:,3*M)
C                           = work array for the SVD routine DSVDC
C           WK(:,3*M+1)    = temporary vector
C           WK(:,3*M+2)    = temporary vector
C           WK(:,3*M+3)    = temporary vector
C           WK(:,3*M+4)    = the saved last column of the matrix
C                            $(W_{\{NKM1\}}^T V_{\{NKM1\}})^{-1}$ 
C           WK(:,3*M+5)    =  $H(NK:NKPl,N)$ 
C           WK(:,3*M+6)    =  $H(NK:NKPl,N+1)$ 
C           WK(:,3*M+7)    = the saved part of  $H(N)$ , used in case
C                           the block is restarted
C           WK(:,3*M+8)    = the saved part of  $H(NPl)$ , used in
C                           case the block is restarted
C           (input/output).
C   Q     = integer array specifying the indices for all the wrapped
C           variables (V,W,SC,WK). To allow the algorithm to run more
C           than M steps, these variable wrap around, in that  $Q(I)$  is
C           the index of the slots where that variables are stored at
C           the I-th step. Normally, these indices would be in order,
C           basically  $Q(I) = I \text{ MOD } M + 1$ , but the algorithm makes no
C           assumptions to this effect. These indices are not checked
C           in any way for validity (input).
C   TOL   = vector with the tolerances used in the various checks. We
C           have:
C           TOL(1) = upper bound for the range of CSI and SIGMA
C           TOL(2) = lower bound for the range of CSI and SIGMA
C           TOL(3) = level below which the norms of the Lanczos
C                   vectors are considered zero (used to check if
C                   an invariant subspace was found)

```

```

C      TOL(4) = level below which the singular values of
C      (W_{NK})^T V_{NK}) are considered zero
C      Note that the scale routine DSCALE expects to receive the
C      tolerances in a 4x1 vector as the one described above. If
C      the user provides non-positive values for any of TOL(1:4)
C      then the DLAL routine will replace them with defaults, as
C      follows:
C      TOL(1) = 1/TOL(2)
C      TOL(2) = eps^{1/2}
C      TOL(3) = eps^{1/4}
C      TOL(4) = eps^{1/3}
C      Here, eps is machine epsilon (input/output).
C      ANORM = user-supplied estimate for the norm of A. If a value less
C      than 2.0 is provided, it is replaced with 2.0. On exit,
C      it is set to the last value used by the algorithm, which
C      updates the norm estimate whenever it is necessary to
C      close a block (input/output).
C      INFO = information passing variable.
C      Upon entry, it gives the numbers of the output units used
C      to trace execution. There are two such units available,
C      one where the non-zero elements of the Hessenberg matrix
C      H are sent, and the other where various trace messages
C      about the progress of the algorithm are sent. The data in
C      H is output in groups of numbers, one number per line, as
C      follows: each group begins with two integers that specify
C      the starting and ending row indices, followed by the
C      actual values, reals output in format E25.18. Note that
C      the ending row index is one higher than the column index,
C      as H is upper Hessenberg. To extract the unit numbers for
C      the two units, INFO = xxyy, where xx is the unit number
C      for the data for H, and yy is the unit number for the
C      trace messages. For example, INFO = 1106 means that the
C      data for H will be sent to unit 11 and the trace messages
C      will be sent to unit 6. A unit number of 00 denotes that
C      no output is to be sent to that unit. INFO = 0 disables
C      both outputs. It is the responsibility of the caller to
C      ensure that the units are ready for output.
C      Upon exit:
C      INFO = 0 ==> nothing to report, algorithm converged
C      INFO < 0 ==> the SVD routine returned this error code
C      in DLAL (with positive sign)
C      INFO = 1 ==> an A-invariant subspace has been found
C      INFO = 2 ==> an A^T-invariant subspace has been found
C      INFO = 3 ==> both subspaces have been found
C      INFO = 4 ==> the last block could not be closed
C      INFO = 8 ==> algorithm failed to converge after NLIM
C      steps
C      INFO = 16 ==> invalid inputs
C      For more details, see the description in the routine DLAL
C      (input/output).
C      H = array dimensioned (NLIM,NLIM) used for H (output).
C
C      External routines used:
C      subroutine dcopy(n,dx,incx,dy,incy)
C      Computes dy = dx.
C      double precision ddot(n,dx,incx,dy,incy)
C      Computes the dot product of dx and dy.
C      subroutine dlal(ndim,nlen,m,n,nk,nkml,vw,sc,wk,q,norms,tol,info)
C      Does one step of the look-ahead Lanczos algorithm.
C      subroutine dscal(n,da,dx,incx).
C      Computes dx = da * dx.
C      subroutine dscale(n,v,w,sc,tol)
C      Scales the Lanczos vectors v and w.
C      subroutine dzero(n,dx,incx)
C      Zeroes out dx.

```

C Noel M. Nachtigal
C October 25, 1990
C

C*****

C INTRINSIC MAX

C EXTERNAL DDOT
C DOUBLE PRECISION DDOT

C INTEGER HDIM, INFO, M, NLEN, NLIM, NDIM, Q(NLIM)
C DOUBLE PRECISION ANORM, H(HDIM,HDIM), SC(5,M), TOL(5)
C DOUBLE PRECISION VW(NDIM,2*M+2), WK(M,3*M+8)

C Local variables.

C INTEGER I, N, NK, NKM1, ONKM1
C INTEGER HF, TF
C DOUBLE PRECISION DTMP, NORMS(2)

C Check whether the inputs are valid.

C IF ((NDIM.LT.1).OR.(NLEN.GT.NDIM).OR.(HDIM.LT.1).OR.(NLIM.GT.HDIM)
C \$.OR.(M.LT.3)) THEN
C INFO = 16
C RETURN
C END IF

C Extract the output units HF and TF from INFO.

C HF = INFO / 100
C TF = INFO - HF * 100

C Initialize the counters.

C N = 1
C NK = 1
C NKM1 = 0

C Scale the first pair of Lanczos vectors.

C SC(4,Q(1)) = 1.0
C SC(5,Q(1)) = 1.0
C SC(3,Q(1)) = 1.0
C CALL DSCALE (NLEN,VW(1,Q(1)),VW(1,M+Q(1)),SC(1,Q(1)),TOL)

C Check for convergence (already?).

C INFO = 0
C IF (SC(4,Q(1)).EQ.-1.0) INFO = INFO + 1
C IF (SC(5,Q(1)).EQ.-1.0) INFO = INFO + 2
C IF (INFO.NE.0) RETURN

C Set up WK(1,1).

C WK(1,1) = DDOT(NLEN,VW(1,Q(1)),1,VW(1,M+Q(1)),1)
C WK(1,1) = SC(5,Q(1)) * SC(4,Q(1)) * WK(1,1)

C Initialize the norm estimate. It has to be at least 2.0.

C DTMP = 2.0
C NORMS(2) = 0.0
C NORMS(1) = MAX(ANORM,DTMP)

C Iterate.
C
C

```

10  ONKM1 = MAX(1,NKM1)
C
C  If we have closed a block, save the working variables, in case we
C  need to restart. Also, reset the norm estimator.
C
  IF (N.EQ.NK) THEN
    CALL DCOPY (N-NKM1+1,WK(1,3*M+6),1,WK(1,3*M+8),1)
    CALL DCOPY (N-ONKM1+1,WK(1,3*M+5),1,WK(1,3*M+7),1)
    NORMS(2) = 0.0
  END IF
C
C  Check whether we have enough room left in the arrays.
C
20  INFO = 0
  IF (N-NKM1+2.GE.M) INFO = 1
  IF ((INFO.NE.0).AND.(TF.NE.0)) THEN
    WRITE (TF,'(A39)') 'Block is maximal, recommending closure.'
  END IF
C
C  Set ANORM to the current value of the norm estimate.
C
  ANORM = NORMS(1)
C
C  Do one step of the Lanczos algorithm.
C
  CALL DLAL (NDIM,NLEN,M,N,NK,NKM1,VW,SC,WK,Q,NORMS,TOL,TF,INFO)
C
C  Check the info passing variable.
C  We check whether the DSVDC routine reported errors or whether the
C  block did not close when it was maximal, both of which result in
C  an immediate return, and whether an invariant subspace was found,
C  which just stops the iteration.
C
  IF (INFO.LT.0) THEN
    NLIM = N
    RETURN
  ELSE IF (INFO.EQ.1) THEN
    NLIM = N
  ELSE IF (INFO.EQ.2) THEN
    NLIM = N
  ELSE IF (INFO.EQ.1 + 2) THEN
    NLIM = N
  ELSE IF (INFO.EQ.4) THEN
    N = NK
    IF (TF.NE.0) WRITE (TF,'(A20)') 'Block did not close:'
C
C  Block did not close, do we have another norm estimate?
C
    IF (NORMS(2).EQ.0.0) THEN
      IF (TF.NE.0) WRITE (TF,'(A47)')
$      '==> no new norm estimates available (aborting).'

```

```

      END IF
      END IF
C
C      Output H to the trace file.
C
      IF (HF.NE.0) THEN
        WRITE (11, '(I20)') ONKM1
        WRITE (11, '(I20)') N
        WRITE (11, '(E25.18)') (WK(I-ONKM1+1,3*M+5), I=ONKM1,N)
      END IF
C
C      Initialize the "next" column of H.  First, zero it all out, then,
C      copy the part we have so far.
C
      CALL DZERO (NLIM,H(1,N-1),1)
      CALL DCOPY (N-ONKM1+1,WK(1,3*M+5),1,H(ONKM1,N-1),1)
C
C      Set up the work vector for the next step.
C
      CALL DCOPY (N-ONKM1+1,WK(1,3*M+6),1,WK(1,3*M+5),1)
C
C      Iterate up to NLIM steps.
C
      IF (N.LT.NLIM) GO TO 10
C
C      Adjust the dimension NLIM, which right now is one bigger than the
C      size of the biggest usable submatrix of H.
C
      NLIM = NLIM - 1
C
      RETURN
      END
C
C*****

```

Copyright (C) 1990, Roland W. Freund and Noel M. Nachtigal
All rights reserved.

No part of this code may be reproduced, stored in a retrieval system, translated, transcribed, transmitted or distributed in any form or by any means means, manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, photocopying, recording, or otherwise, without the prior explicit written permission of the author(s) or their designated proxies. In no event shall the above copyright notice be removed or altered in any way.

This code is provided "as is", without any warranty of any kind, either expressed or implied, including but not limited to, any implied warranty of merchantability or fitness for any purpose. In no event will any party who distributed the code be liable for damages or for any claim(s) by any other party, including but not limited to, any lost profits, lost monies, lost data or data rendered inaccurate, losses sustained by third parties, or any other special, incidental or consequential damages arising out of the use or inability to use the program, even if the possibility of such damages has been advised against. The entire risk as to the quality, the performance, and the fitness of the program for any particular purpose lies with the party using the code.

ANY USE OF THIS CODE CONSTITUTES ACCEPTANCE OF THE TERMS OF THE
ABOVE STATEMENTS

This file contains the routines for the QMR algorithm. SYSLAL is the basic routine used to solve linear systems with QMR. BCKSUB is a triangular matrix solver geared towards the setup used by the Lanczos code (in particular, vector wrapping).

SUBROUTINE SYSLAL (NDIM,NLEN,NLIM,M,VW,SC,WK,Q,TOL,ANORM,INFO)

Purpose:

This subroutine uses the Lanczos algorithm, combined with the QMR algorithm, to solve linear systems. It runs the QMR algorithm to convergence or until a user-specified iteration limit is reached. The caller initializes the following:

VW(:,Q(1)) = the first Lanczos vector V_1, the residual for the initial guess
VW(:,M+Q(1)) = the first Lanczos vector W_1
Q = the array of wrapped indices
TOL(1:4) = tolerances for the Lanczos algorithm (optional)
TOL(5) = convergence tolerance for the residual norm
ANORM = estimate for the norm of the matrix (optional)
INFO = the output units, if any

If the user provides non-positive values for the tolerances in TOL(1:4), the Lanczos routine DLAL will supply its own defaults. Also, if the user provides a norm estimate of less than 2.0, the Lanczos routine DLAL will set the norm estimate to 2.0.

Parameters:

NDIM = the dimensioned size of the array VW. Must be at least 1; checked for validity (input).
NLEN = the actual size of the Lanczos vectors V and W; this also implicitly determines the size of the matrix A. Must be

```

C      less than or equal to NDIM; checked for validity (input).
C      NLIM - the maximum number of iterations the algorithm can take.
C            Must be at least 1; checked for validity. On exit, it is
C            the index of the last step taken or attempted, depending
C            on INFO, see below (input/output).
C      M      - the maximum number of Lanczos vectors that can be stored
C            in the array VW. It is related to the size of the largest
C            block that can be built. The algorithm runs out of memory
C            when the number of vectors in the last block and in the
C            current block reaches M. For this reason, M must be at
C            least 3; checked for validity (input).
C      VW      - work array dimensioned (NDIM,3*M+4) words. It is used to
C            store the Lanczos vectors V in VW(:,1:M), the vectors W
C            in VW(:,M+1:2*M), two temporary vectors used by DLAL in
C            VW(:,2*M+1) and VW(:,2*M+2), the right hand side vector B
C            in VW(:,2*M+3), the current solution X_N in VW(:,2*M+4),
C            and the update direction vectors PR in VW(:,2*M+5:3*M+4).
C            The Lanczos vectors V and W and the direction vectors PR
C            are stored wrapped, i.e., V_N is stored in VW(:,Q(N)) and
C            W_N is stored in VW(:,M+Q(N)), where Q(N) is assumed to
C            be a wrapped index array -- see below (input/output).
C      SC      - work array dimensioned (5,M), used to store the various
C            scale factors. We have:
C            SC(1,i) = S(i) / S(i-1)
C            SC(2,i) = T(i) / T(i-1)
C            SC(3,i) = S(i) / T(i)
C            SC(4,i) = CSI(i)
C            SC(5,i) = SIG(i)
C            Note that the scale routine DSCALE expects to receive the
C            scale factors in a 5x1 vector as the one described above.
C            This routine initializes the first column; thereafter,
C            the DLAL routine will update the array (input/output).
C      WK      - work array dimensioned (M,5*M+14), used to store internal
C            variables. We have:
C            WK(:,1:M) = (W_{NK})^T V_{NK})
C            WK(:,M+1) to WK(:,2*M)
C            = (W_{NK})^T V_{NK})^{-1}
C            WK(:,2*M+1) to WK(:,3*M)
C            = work array for the SVD routine DSVDC
C            WK(:,3*M+1) = temporary vector
C            WK(:,3*M+2) = temporary vector
C            WK(:,3*M+3) = temporary vector
C            WK(:,3*M+4) = the saved last column of the matrix
C            (W_{NKM1})^T V_{NKM1})^{-1}
C            WK(:,3*M+5) = H(NK:NKP1,N)
C            WK(:,3*M+6) = H(NK:NKP1,N+1)
C            WK(:,3*M+7) = the saved part of H(N), used in case
C            the block is restarted
C            WK(:,3*M+8) = the saved part of H(NP1), used in
C            case the block is restarted
C            WK(:,3*M+9) = the scale factors OMEGA (wrapped)
C            WK(:,3*M+10) = the cosines of the Givens rotations
C            (wrapped)
C            WK(:,3*M+11) = the sines of the Givens rotations
C            (wrapped)
C            WK(:,3*M+12) = the rotated right hand side for the
C            least squares problem (wrapped)
C            WK(:,3*M+13) = the elements of the row vectors ZNK
C            (wrapped)
C            WK(:,3*M+14) = the last column of R_{NKM2})^{-1}
C            (wrapped)
C            WK(:,3*M+15) to WK(:,4*M+14)
C            = the matrix Y_NK (wrapped)
C            WK(:,4*M+15) to WK(:,5*M+14)
C            = the matrix R_NK (wrapped)
C            (input/output).

```

Q - integer array specifying the indices for all the wrapped variables (V,W,SC,WK). To allow the algorithm to run more than M steps, these variable wrap around, in that Q(I) is the index of the slots where that variables are stored at the I-th step. Normally, these indices would be in order, basically $Q(I) = I \text{ MOD } M + 1$, but the algorithm makes no assumptions to this effect. These indices are not checked in any way for validity (input).

TOL - vector with the tolerances used in the various checks. We have:

TOL(1) = upper bound for the range of CSI and SIGMA
TOL(2) = lower bound for the range of CSI and SIGMA
TOL(3) = level below which the norms of the Lanczos vectors are considered zero (used to check if an invariant subspace was found)
TOL(4) = level below which the singular values of $(W_{\{NK\}}^T V_{\{NK\}})$ are considered zero
TOL(5) = the convergence level for the scaled residual norm

Note that the scale routine DSCALE expects to receive the tolerances in a 4x1 vector as the one described above. If the user provides non-positive values for any of TOL(1:4) then the DLAL routine will replace them with defaults, as follows:

TOL(1) = 1/TOL(2)
TOL(2) = $\text{eps}^{\{1/2\}}$
TOL(3) = $\text{eps}^{\{1/4\}}$
TOL(4) = $\text{eps}^{\{1/3\}}$

Here, eps is machine epsilon (input/output).

ANORM - user-supplied estimate for the norm of A. If a value less than 2.0 is provided, it is replaced with 2.0. On exit, it is set to the last value used by the algorithm, which updates the norm estimate whenever it is necessary to close a block (input/output).

INFO - information passing variable.

Upon entry, it gives the numbers of the output units used to trace execution, and controls the generation of the convergence history. Up to three trace output units can be specified: one where the non-zero elements of the upper Hessenberg matrix H are sent, another where various trace messages with details about the progress of the algorithm are sent, and finally a third one, where data about the convergence history is sent. The data in H is output in groups of numbers, one number per line, as follows: each group begins with two integers that specify the starting and ending row indices, followed by the actual values, reals output in format E25.17. Note that the ending row index is one higher than the column index, as H is upper Hessenberg. Also note that since H is output before the block is checked for forced closure, data for any columns in H may appear several times; of course, only the latest is valid. The convergence history data is output as up to three numbers on a line, the first one an integer, the size of the current block, the other two reals, output in format E25.17, the first one if which is the upper bound used in the convergence tests, and the second one is the computed residual norm, when actually computed, or -1.0 otherwise. To extract the unit numbers for these units, INFO = txxyyzz, where xx is the unit number for the data for H, and yy is the unit number for the convergence data and zz of the unit number for the trace messages. Also, if t.NE.0, then the actual residual norm is computed at every step. For example:

INFO = 1106 ==> trace messages are sent to unit 6,
convergence data sent to unit 11

INFO = 1000000 ==> the true residual norm is always

```

INFO - 5121314 --> computed
                    always compute true residual norm,
                    send trace messages to unit 14, the
                    convergence data to unit 13, and H
                    to unit 12

```

INFO = 0 ==> no output.

Note that INFO must be a 32-bit integer, and that it is the responsibility of the caller to ensure that the units used are ready for output.

Upon exit:

INFO = 0 ==> nothing to report, algorithm converged

```
INFO < 0  --> the SVD routine returned this error code
              in DLAL (with positive sign)
```

INFO = 1 --> an A-invariant subspace has been found

INFO = 2 --> an A^T-invariant subspace has been found

INFO = 3 --> both subspaces have been found

INFO - 4 ==> the last block could not be closed

```
INFO - 8  --> algorithm failed to converge after NLIM
           steps
```

INFO = 16 ==> invalid inputs

For more details, see the description in the routine DLAL (input/output).

External routines used:

```
subroutine axb(x,b)
```

Computes $b = A * x$.

```
subroutine bcksub(ndim,nlen,nstart,a,xb,q)
```

Computes $xb = \text{inv}(a) * xb$ with a upper triangular (specific to α setup, i.e., the columns of a are permuted according to q).

```
subroutine daxpy(n,da,dx,incx,dy,incy)
```

Computes $dy = da * dx + dy$.

```
subroutine dcopy(n,dx,incx,dy,incy)
```

Computes $dy = dx$.

```
double precision ddot (n,dx,incx,dy,incy)
```

Computes the dot product of dx and dy.

```
subroutine dlad(ndim,nlen,m,n,nk,nkml,vw,sc,wk,q,norms,tol,info)
```

Does one step of the look-ahead Lanczos algorithm.

```
double precision dnorm2(n,dx,incx)
```

Computes the 2-norm of dx.

```
subroutine drot (n,dx,incx,dy,incy,dcos,dsin)
```

Applies a Givens rotation to a vector.

```
subroutine drotg(da,db,dcos,dsin)
```

Computes a Givens rotation.

```
subroutine dscal(n,da,dx,incx).
```

```
Computes dx = da * dx.
```

```
subroutine dscale(n,v,w,sc,tol)
```

Scales the Lanczos vectors v and w .

```
subroutine dzero(n,dx,incx)
```

Zeroes out dx.

```
double precision getomg(n)
```

Computes the scaling factor OMEGA n.

Noel M. Nachtigal

October 24, 1990

INTRINSIC FLOAT, MAX, SORT

EXTERNAL DDOT, DNRM2, GETOMG

DOUBLE PRECISION DDOT, DNRM2, GETOMG

```
INTEGER INFO, M, NLEN, NLIM, NDIM, Q(NLIM)
```

DOUBLE PRECISION ANORM, SC(5,M), TOL(5)

DOUBLE PRECISION VW (NDIM, 3*M+4), WK (M, 5*M+14)

```

C      Local variables.
C
      INTEGER I, J, LNK, LNKM1, LNKM2, N, NK, NKM1, ONK, ONKM1, ONKM2
      INTEGER HBASE, HF, TF, TRES, VF
      DOUBLE PRECISION DTMP1, DTMP2, MAXOMG, NORMS(2), R0, SAVRHS
C
C      Check whether the inputs are valid.
C
      IF ((NDIM.LT.1).OR.(NLEN.GT.NDIM).OR.(NLIM.LT.1).OR.(M.LT.3)) THEN
          INFO = 16
          RETURN
      END IF
C
C      Extract the output units HF, TF, and VF from INFO, and the true
C      residual flag TRES.
C
      TRES = INFO / 1000000
      INFO = INFO - TRES * 1000000
      HF = INFO / 10000
      INFO = INFO - HF * 10000
      TF = INFO / 100
      INFO = INFO - TF * 100
      VF = INFO
C
C      Initialize the counters.
C
      N = 1
      NK = 1
      NKM1 = 0
C
C      Scale the first pair of Lanczos vectors.
C
      SC(4,Q(1)) = 1.0
      SC(5,Q(1)) = 1.0
      SC(3,Q(1)) = 1.0
      CALL DSCALE (NLEN,VW(1,Q(1)),VW(1,M+Q(1)),SC(1,Q(1)),TOL)
C
C      Check for convergence (already?).
C
      INFO = 0
      IF (SC(4,Q(1)).EQ.-1.0) INFO = INFO + 1
      IF (SC(5,Q(1)).EQ.-1.0) INFO = INFO + 2
      IF (INFO.NE.0) RETURN
C
C      Set up WK(1,1).
C
      WK(1,1) = DDOT(NLEN,VW(1,Q(1)),1,VW(1,M+Q(1)),1)
      WK(1,1) = SC(5,Q(1)) * SC(4,Q(1)) * WK(1,1)
C
C      Set up the first element of the right-hand side.
C
      WK(Q(1),3*M+9) = GETOMG(1)
      WK(Q(1),3*M+12) = WK(Q(1),3*M+9) * SC(1,Q(1))
      MAXOMG = 1.0 / WK(Q(1),3*M+9)
C
C      Initialize the norm estimate. It has to be at least 2.0.
C
      DTMP1 = 2.0
      NORMS(2) = 0.0
      NORMS(1) = MAX(ANORM,DTMP1)
C
C      Compute and save the initial residual norm in R0.
C
      R0 = DNRM2(NLEN,VW(1,Q(1)),1)
C
C      Iterate.

```

```

C
10  ONK   = NK
    ONKM1 = MAX(1,NKM1)
C
C    If we have closed a block, save the working variables, in case we
C    need to restart. Also, reset the norm estimator.
C
    IF (N.EQ.NK) THEN
        CALL DCOPY (N-ONKM1+1,WK(1,3*M+6),1,WK(1,3*M+8),1)
        CALL DCOPY (N-ONKM1+1,WK(1,3*M+5),1,WK(1,3*M+7),1)
        SAVRHS   = WK(Q(N),3*M+12)
        NORMS(2) = 0.0
    END IF
C
C    Check whether we have enough room left in the arrays.
C
20  INFO = 0
    IF (N-ONKM1+2.GE.M) INFO = 1
    IF ((INFO.NE.0).AND.(VF.NE.0)) THEN
        WRITE (VF,'(A39)') 'Block is maximal, recommending closure.'
    END IF
C
C    Set ANORM to the current value of the norm estimate.
C
    ANORM = NORMS(1)
C
C    Do one step of the Lanczos algorithm.
C
    CALL DLAL (NDIM,NLEN,M,N,NK,NKM1,VW,SC,WK,Q,NORMS,TOL,VF,INFO)
C
C    Check the info passing variable.
C    We check whether the DSVDC routine reported errors, whether the
C    block did not close when it was maximal, or whether an invariant
C    subspace was found.
C
    IF (INFO.LT.0) THEN
        NLIM = N
        RETURN
    ELSE IF (INFO.EQ.1) THEN
        NLIM = N
        RETURN
    ELSE IF (INFO.EQ.2) THEN
        NLIM = N
        RETURN
    ELSE IF (INFO.EQ.1 + 2) THEN
        NLIM = N
        RETURN
    ELSE IF (INFO.EQ.4) THEN
        N = NK
        IF (VF.NE.0) WRITE (VF,'(A20)') 'Block did not close:'
C
C    Block did not close, do we have another norm estimate?
C
        IF (NORMS(2).EQ.0.0) THEN
            IF (VF.NE.0) WRITE (VF,'(A47)')
$            '==> no new norm estimates available (aborting).'
            NLIM = N
            RETURN
        ELSE
C
C    Update the norm --- the block is guaranteed to close now. We then
C    restart the block.
C
            NORMS(2) = 2.0 * NORMS(2)
            IF (VF.NE.0) WRITE (VF,'(A30, E25.17)')
$            '==> updating norm estimate to ', NORMS(2)

```

```

        CALL DCOPY (N-ONKM1+1,WK(1,3*M+8),1,WK(1,3*M+6),1)
        CALL DCOPY (N-ONKM1+1,WK(1,3*M+7),1,WK(1,3*M+5),1)
        NORMS(1) = NORMS(2)
        WK(Q(N),3*M+12) = SAVRHS
        NORMS(2) = 0.0
        GO TO 20
    END IF
ELSE
    INFO = 8
END IF

C
C
C
Output H to the trace file.

IF (HF.NE.0) THEN
    WRITE (11,'(I20)') ONKM1
    WRITE (11,'(I20)') N
    WRITE (11,'(E25.17)') (WK(I-ONKM1+1,3*M+5),I=ONKM1,N)
END IF

C
C
C
Get the next scaling factor OMEGA(NP1) and update MAXOMG.

WK(Q(N),3*M+9) = GETOMG(N)
MAXOMG = MAX(MAXOMG,1.0/WK(Q(N),3*M+9))

C
C
C
Compute the starting index in H(:,N-1).

HBASE = MAX(1,ONKM1-1)

C
C
C
Multiply the column of H by the OMEGAs and apply all the previous
rotations.

WK(1,3*M+5) = WK(Q(HBASE),3*M+9) * WK(1,3*M+5)

C
C
C
If we are beyond the first block, then there is fill-in above the
top element in this column of H. This element is saved in ZNK.

C
IF (ONKM1.GT.1) THEN
    DTMP1 = 0.0
    CALL DROT (1,DTMP1,1,WK(1,3*M+5),1,
$      WK(Q(ONKM1),3*M+10),WK(Q(ONKM1),3*M+11))
    WK(Q(N-1),3*M+13) = DTMP1
END IF

C
C
C
Apply the rotations to the rest of the column.

DO 30 I = ONKM1+1, N-1
    J = I - ONKM1
    WK(J,3*M+5) = WK(Q(I),3*M+9) * WK(J,3*M+5)
    CALL DROT (1,WK(J,3*M+5),1,WK(J+1,3*M+5),1,
$      WK(Q(I),3*M+10),WK(Q(I),3*M+11))
30 CONTINUE

C
C
C
H(N) is not reached by the any of the rotations. Multiply it by
its OMEGA and compute the rotation for the column. This will also
apply it.

C
J = N - ONKM1
WK(J,3*M+5) = WK(Q(N),3*M+9) * WK(J,3*M+5)
CALL DROTG (WK(J,3*M+5),WK(J+1,3*M+5),
$      WK(Q(N),3*M+10),WK(Q(N),3*M+11))

C
C
C
Apply it to the right-hand side as well.

WK(Q(N),3*M+12) = 0.0
CALL DROT (1,WK(Q(N-1),3*M+12),1,WK(Q(N),3*M+12),1,
$      WK(Q(N),3*M+10),WK(Q(N),3*M+11))

```

```

C      Extract the next column of Y_{NK}.
C
C      IF (ONK.GT.ONKM1) THEN
C          CALL DCOPY (ONK-ONKM1,WK(1,3*M+5),1,WK(1,3*M+14+Q(N-1)),1)
C      END IF
C
C      Extract the next column of R_{NK}.
C
C      CALL DCOPY (N-ONK,WK(ONK-ONKM1+1,3*M+5),1,WK(1,4*M+14+Q(N-1)),1)
C
C      Set up the work vector for the next step.
C
C      CALL DCOPY (N-ONKM1+1,WK(1,3*M+6),1,WK(1,3*M+5),1)
C
C      If we have closed a block, update the solution.
C
C      IF (NK.NE.ONK) THEN
C
C          Compute the lengths of the blocks.
C
C          LNK   = N - ONK
C          LNKM1 = ONK - ONKM1
C          LNKM2 = ONKM1 - ONKM2
C
C          Zero out P_{NK} R_{NK}.
C
C          DO 40 I = ONK, N-1
C              CALL DZERO (NLEN,VW(1,2*M+4+Q(I)),1)
40      CONTINUE
C
C          IF (ONKM1.GT.1) THEN
C
C              Compute the term involving P_{NKM2}.
C
C              IF (LNKM2.EQ.1) THEN
C
C                  LNKM2 = 1 ==> R_{NKM2}^{-1} is a 1x1 matrix, so we first multiply
C                  ZNK by this scalar, then ( P_{NKM2} R_{NKM2} ) by the result.
C
C                  DTMP1 = WK(Q(ONKM1-1),3*M+14)
C                  DO 50 I = ONK,N-1
C                      DTMP2 = DTMP1 * WK(Q(I),3*M+13)
C                      CALL DCOPY (NLEN,VW(1,2*M+4+Q(ONKM1-1)),1,
C                      $          VW(1,2*M+4+Q(I)),1)
C                      CALL DSCAL (NLEN,-DTMP2,VW(1,2*M+4+Q(I)),1)
50      CONTINUE
C
C                  ELSE IF (LNK.EQ.1) THEN
C
C                      LNK = 1 ==> ZNK is a 1x1 vector, so we first multiply the last
C                      column of R_{NKM2}^{-1} by ZNK first, then ( P_{NKM2} R_{NKM2} )
C                      by the result.
C
C                      DTMP1 = WK(Q(ONK),3*M+13)
C                      DO 60 I = ONKM2,ONKM1-1
C                          DTMP2 = DTMP1 * WK(Q(I),3*M+14)
C                          CALL DAXPY (NLEN,-DTMP2,VW(1,2*M+4+Q(I)),1,
C                          $          VW(1,2*M+4+Q(ONK)),1)
60      CONTINUE
C
C                      ELSE
C
C                          Otherwise, we first multiply ( P_{NKM2} R_{NKM2} ) by the last
C                          column of R_{NKM2}^{-1}, accumulating the result in the first
C                          column of ( P_{NK} R_{NK} ), which was zeroed out.

```

```

C      DO 70 I = ONKM2, ONKM1-1
C          CALL DAXPY (NLEN, WK(Q(I), 3*M+14), VW(1, 2*M+4+Q(I)), 1,
$              VW(1, 2*M+4+Q(ONK)), 1)
70      CONTINUE
C
C      Then multiply by the ZNK vector, which involves just scaling the
C      above column by the appropriate factor and storing it in its slot
C      in the array.
C
C          DO 80 I = ONK+1, N-1
C              CALL DCOPY (NLEN, VW(1, 2*M+4+Q(ONK)), 1,
$                  VW(1, 2*M+4+Q(I)), 1)
C              CALL DSCAL (NLEN, -WK(Q(I), 3*M+13), VW(1, 2*M+4+Q(I)), 1)
80      CONTINUE
C          CALL DSCAL (NLEN, -WK(Q(ONK), 3*M+13),
$              VW(1, 2*M+4+Q(ONK)), 1)
C          END IF
C          END IF
C
C          IF (ONK.GT.ONKM1) THEN
C
C              Compute the term involving P_{NKM1}.
C
C              DO 100 I = ONK, N-1
C
C                  Compute the next column of R_{NKM1}^{-1} * Y_{NK}.
C
C                      CALL BCKSUB (M, LNKM1, ONKM1, WK(1, 4*M+15),
$                          WK(1, 3*M+14+Q(I)), Q)
C
C                  Multiply ( P_{NKM1} R_{NKM1} ) by the new column and add to the
C                  appropriate column of ( P_{NK} R_{NK} ).
C
C                      DO 90 J = 1, LNKM1
C                          CALL DAXPY (NLEN, -WK(J, 3*M+14+Q(I)),
$                              VW(1, 2*M+4+Q(ONKM1+J-1)), 1, VW(1, 2*M+4+Q(I)), 1)
90                      CONTINUE
100                     CONTINUE
C
C                  Compute the last column of R_{NKM1} and store it for when it will
C                  become the last column of R_{NKM2}.
C
C                      CALL DZERO (M, WK(1, 3*M+14+Q(ONK)), 1)
C                      WK(LNKM1, 3*M+14+Q(ONK)) = 1.0
C                      CALL BCKSUB (M, LNKM1, ONKM1, WK(1, 4*M+15),
$                          WK(1, 3*M+14+Q(ONK)), Q)
C                      DO 110 I = ONKM1, ONK-1
C                          WK(Q(I), 3*M+14) = WK(I-ONKM1+1, 3*M+14+Q(ONK))
110                     CONTINUE
C                      END IF
C
C              Now add V_{NK} into ( P_{NK} R_{NK} ). We also use this loop to
C              copy the right hand side of the least squares problem to a work
C              vector.
C
C              DO 120 I = ONK, N-1
C                  CALL DAXPY (NLEN, SC(5, Q(I)), VW(1, Q(I)), 1,
$                      VW(1, 2*M+4+Q(I)), 1)
C                  WK(I-ONK+1, 3*M+14+Q(ONK)) = WK(Q(I), 3*M+12)
120             CONTINUE
C
C              Compute R_{NK}^{-1} * RHS.
C
C                  CALL BCKSUB (M, LNK, ONK, WK(1, 4*M+15), WK(1, 3*M+14+Q(ONK)), Q)

```

```

C
C      Update the solution vector.
C
      DO 130 I = ONK, N-1
          CALL DAXPY (NLEN,WK(I-ONK+1,3*M+14+Q(ONK)),
$          VW(1,2*M+4+Q(I)),1,VW(1,2*M+4),1)
130  CONTINUE
C
C      Compute the residual norm upper bound.
C
      DTMP1 = SQRT(FLOAT(N)) * MAXOMG * ABS(WK(Q(N),3*M+12)) / R0
      IF (VF.NE.0) WRITE (VF,'(A30,E25.17)') 'Upper bound:', DTMP1
C
C      If the upper bound is within one order of magnitude of the target
C      convergence norm, compute the actual scaled residual norm.
C
      IF ((TRES.NE.0).OR.(DTMP1/TOL(5).LE.10.0)) THEN
          DTMP2 = DTMP1
          CALL AXB (VW(1,2*M+4),VW(1,2*M+1))
          DTMP1 = 1.0
          CALL DSCAL (NLEN,-DTMP1,VW(1,2*M+1),1)
          CALL DAXPY (NLEN,DTMP1,VW(1,2*M+3),1,VW(1,2*M+1),1)
          DTMP1 = DNRM2(NLEN,VW(1,2*M+1),1) / R0
          IF (VF.NE.0) WRITE (VF,'(A30,E25.17)') 'Actual norm:', DTMP1
          IF (TF.NE.0) WRITE (TF,'(I10,2E25.17)') N-ONK, DTMP2, DTMP1
      ELSE
          IF (TF.NE.0) WRITE (TF,'(I10,2E25.17)') N-ONK, DTMP1, -1.0
      END IF
C
C      Check for convergence.
C
      IF (DTMP1.LE.TOL(5)) THEN
          INFO = 0
          NLIM = N
      END IF
C
C      Update ONKM2.
C
      ONKM2 = ONKM1
      END IF
C
C      Iterate up to NLIM steps.
C
      IF (N.LT.NLIM) GO TO 10
C
      RETURN
      END
C
C*****
C
      SUBROUTINE BCKSUB (NDIM,NLEN,NSTART,A,XB,Q)
C
C      Purpose:
C      Computes  $XB = \text{inv}(A) * XB$ , where A is upper triangular.
C
C      Parameters:
C      NDIM    = the dimensioned size of A (input).
C      NLEN    = the actual size of A (input).
C      NSTART  = the starting column index for A (input).
C      A       = array containing the matrix of interest starting in the
C               column NSTART and possibly wrapping around as indicated
C               by Q (input).
C      XB      = upon entry, the right hand side vector; upon exit, the
C               solution. XB is not wrapped (input/output).
C      Q       = integer array specifying the actual indices for wrapping
C               purposes. Q(i) is the true index in A of the ith column

```

```

C           of the matrix of interest (input).
C
C      Noel M. Nachtigal
C      October 8, 1990
C
C      INTEGER NDIM, NLEN, NSTART, Q(*)
C      DOUBLE PRECISION A(NDIM,*), XB(*)
C
C      Local variables.
C
C      INTEGER I, ILAST, ITMP, J
C      DOUBLE PRECISION DTMP
C
C      Do the elimination; the only trick here is to keep track of the
C      wrapped columns of A, i.e., A(:,J) is stored in A(:,Q(J)).
C
C      ILAST = NSTART + NLEN - 1
C      XB(NLEN) = XB(NLEN) / A(NLEN,Q(ILAST))
C      DO 20 I = ILAST-1, NSTART, -1
C          DTMP = 0.0
C          ITMP = I - NSTART + 1
C          DO 10 J = I+1, ILAST
C              DTMP = DTMP + XB(J-NSTART+1) * A(ITMP,Q(J))
10      CONTINUE
C          XB(ITMP) = (XB(ITMP) - DTMP) / A(ITMP,Q(I))
20  CONTINUE
C
C      RETURN
C      END
C
C*****

```

```

C*****
C
C Copyright (C) 1990, Roland W. Freund and Noel M. Nachtigal
C All rights reserved.
C
C No part of this code may be reproduced, stored in a retrieval
C system, translated, transcribed, transmitted or distributed in
C any form or by any means means, manual, electric, electronic,
C electro-magnetic, mechanical, chemical, optical, photocopying,
C recording, or otherwise, without the prior explicit written
C permission of the author(s) or their designated proxies. In no
C event shall the above copyright notice be removed or altered in
C any way.
C
C This code is provided "as is", without any warranty of any kind,
C either expressed or implied, including but not limited to, any
C implied warranty of merchantability or fitness for any purpose.
C In no event will any party who distributed the code be liable for
C damages or for any claim(s) by any other party, including but not
C limited to, any lost profits, lost monies, lost data or data
C rendered inaccurate, losses sustained by third parties, or any
C other special, incidental or consequential damages arising out of
C the use or inability to use the program, even if the possibility
C of such damages has been advised against. The entire risk as to
C the quality, the performance, and the fitness of the program for
C any particular purpose lies with the party using the code.
C
C *****
C ANY USE OF THIS CODE CONSTITUTES ACCEPTANCE OF THE TERMS OF THE
C ABOVE STATEMENTS
C *****
C*****
C
C This file contains the coefficient functions used by the Lanczos
C algorithm in the recursion formulas for the inner vectors. The
C basic recursions are of the form:
C
C      
$$V_{N+1} = A * V_N - ZETA_N * V_N - ETA_N * V_{N-1}$$

C      
$$W_{N+1} = A^T * W_N - ZETA_N * W_N - ETA_N * W_{N-1}$$

C
C The functions in this file compute the coefficients ZETA_N and
C ETA_N for the various indices N.
C*****
C
C DOUBLE PRECISION FUNCTION DETA(I)
C
C Purpose:
C Returns the second scalar in the recursion used for inner vectors
C \theta_{i+1} = ( z - dzeta(i) ) \theta_i - deta(i) \theta_{i-1}.
C
C Parameters:
C I = the degree of the current polynomial, see above (input).
C
C Noel M. Nachtigal
C August 28, 1990
C
C INTEGER I
C
C DETA = 1.0
C
C RETURN
C END
C*****

```

```

C      DOUBLE PRECISION FUNCTION DZETA(I)
C
C      Purpose:
C      Returns the first scalar in the recursion used for inner vectors,
C      \theta_{i+1} = ( z - dzeta(i) ) \theta_i - deta(i) \theta_{i-1}.
C
C      Parameters:
C      I = the degree of the current polynomial, as above (input).
C
C      Noel M. Nachtigal
C      August 28, 1990
C
C      INTEGER I
C
C      DZETA = 1.0
C
C      RETURN
C      END
C*****

```

```

C*****
C
C Copyright (C) 1990, Roland W. Freund and Noel M. Nachtigal
C All rights reserved.
C
C No part of this code may be reproduced, stored in a retrieval
C system, translated, transcribed, transmitted or distributed in
C any form or by any means means, manual, electric, electronic,
C electro-magnetic, mechanical, chemical, optical, photocopying,
C recording, or otherwise, without the prior explicit written
C permission of the author(s) or their designated proxies. In no
C event shall the above copyright notice be removed or altered in
C any way.
C
C This code is provided "as is", without any warranty of any kind,
C either expressed or implied, including but not limited to, any
C implied warranty of merchantability or fitness for any purpose.
C In no event will any party who distributed the code be liable for
C damages or for any claim(s) by any other party, including but not
C limited to, any lost profits, lost monies, lost data or data
C rendered inaccurate, losses sustained by third parties, or any
C other special, incidental or consequential damages arising out of
C the use or inability to use the program, even if the possibility
C of such damages has been advised against. The entire risk as to
C the quality, the performance, and the fitness of the program for
C any particular purpose lies with the party using the code.
C
C *****
C ANY USE OF THIS CODE CONSTITUTES ACCEPTANCE OF THE TERMS OF THE
C ABOVE STATEMENTS
C *****
C*****
C
C This file contains the scaling function GETOMG, which computes
C the scaling factors Omega used in scaling the Hessenberg matrix
C from the least squares problem solved by QMR.
C*****
C
C DOUBLE PRECISION FUNCTION GETOMG (I)
C
C Purpose:
C Returns the scaling parameter OMEGA(i).
C
C Parameters:
C I = the index of the parameter OMEGA (input).
C
C Noel M. Nachtigal
C October 7, 1990
C
C INTEGER I
C
C GETOMG = 1.0
C
C RETURN
C END
C*****

```